Coláiste na Tríonóide, Baile Átha Cliath
Trinity College Dublin
Ollscoil Átha Cliath | The University of Dublin

Self Tuning Algorithms Running on Intel's Transactional Memory

— An investigation into Self Tuning Split Transaction algorithms running on Intel's Hardware Transactional Memory —

DANIEL DOWD

Supervisor: Jeremy Jones, Head of the Department of Computer Science and Statistics

Examiner: David Gregg, Department of Computer Science and Statistics

Submitted to the University of Dublin, Trinity College in order to fulfil the requirements for the award of the MAI in Computer Engineering

May, 2016

Declaration

I, Daniel Dowd, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signature Date

Permission To Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

Signature

Date

Abstract

As processors grow into a new era where the number of cores on a chip is growing to meet consumer demands for greater performance, we need to utilise new advancements in hardware to meet these demands. Intel have recently released their implementation of Hardware Transactional Memory and using this it is possible to obtain greater performance of multithreaded applications. However, serial data structures do not benefit from these new advancements in the same way as more parallel data structures such as a binary search tree do. Using a technique called Split Transactions it is possible to show considerable performance increases over current state of the art approaches with a serial data structure such as an ordered linked list. The performance increase that is observed by using this method on an ordered linked list can be up to 9 times greater than transactional memory which is not utilising split transactions. Split transactions are also 9 times faster than a conventional lock based approach.

Summary

The recent move of processors towards more and more cores creates performance bottlenecks on multithreaded applications that require locks. This limits the performance of multithreaded applications as all but one thread, the thread that has obtained the lock, are not progressing. Hardware Transactional Memory provides a solution to this problem. It allows all threads to make speculative changes and only commit the changes it has made if there are no conflicts with the speculative changes of other threads.

Unfortunately Intel's implementation of Hardware Transactional Memory needs a lock based fallback path. A transaction is never guaranteed to be able to commit its changes. A transaction abort can be caused by events other than the transaction going into a conflict status. A page fault is an example of this. Some errors such as a page fault can not be handled by a transaction and this is why there needs to be a lock based fallback path.

This has been shown to work well with parallel data structures but, it does have limitations when it comes to serial data structures. Split transactions provide a method of splitting up a long transaction into a number of smaller transactions. Split transactions carry more overhead costs in setting them up because there are more of them happening for one operation on a data structure to complete but, they can drastically reduce the likelihood of a conflict between two threads. This can reduce the number of aborts

that occur and this reduces the number of times the fallback path is taken.

The likelihood that a transaction will need to take the fallback path, when split transactions are being used, can be predicted. Using the birthday problem algorithm as a base, this can be adapted to give the probability that two threads will be in the same split at the same time. This adaptation can be used in conjunction with the probability that a split transaction will operate in a split to find the percentage of operations that will require a lock to complete.

The split length of a split transaction is crucial to a the split transaction committing its portion of the work. If the split length is too long, the fallback path will need to be taken too often, while if the split length is too short, the high overhead of setting up a new transaction is going to negatively impact performance. This ideal split length value will change with the key range of a list and the ideal length could even change within the course of the benchmark test to check its performance. A static analysis was performed to find the upper and lower bounds for the split lengths. This was integrated into the dynamic tuning algorithm that was developed to tune the split length on the fly.

The performance of the dynamically tuned split transaction was compared against transactional memory that was not running split transactions and a lock. All tests were run on ordered linked lists with key ranges of 64, 4096(4K) and 65536(64K). The lock used in this comparison was the same type of lock that was used in the fallback path of the transactions. This lock was the Test and Test and Set lock. It was chosen for its speed and efficiency of bus usage.

Acknowledgements

I would like to thank Jeremy Jones for the time and effort he invested in me. He was always willing to answer my questions and give his support in whichever way he could throughout the project. I would also like to thank Cian Burns for giving his feedback throughout the writing of my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Problem

Up until 2007 Moore's Law [Moore, 2006] held in terms of computers. It stated that every 18 months the clock frequency of computers would have doubled, thereby doubling the speed of the computer. After 2007 this law no longer held. This led to a change in the approach of processor design. Instead of the single or dual core processors that were popular up until 2007, the processors began to contain 4 or more cores. Up to this point computers with this number of cores were almost exclusively super-computers, not the consumer grade readily available chips that we now see.

Multicore processors can increase the performance of computers as multiple tasks can run at the same time on different cores. However, when using a shared data structure this is not always the case. When two threads are running and attempting to operate on the same data structure, only one can do so at any point in time. If this does not happen it is possible that one of the threads will change data that the other thread was using without its knowledge. This results in the data structure becoming corrupt. In order to prevent this the data structure must be protected by a lock or a mutual

exclusion algorithm. This ensures that only one thread at a time can operate on the data structure. This can lead to a large amount of idle cores which is a waste of resources. If a processor has eight cores with a program that has eight threads running on that processor, all trying to update a shared data structure, the program can at best achieve 1/8th of the amount of work that it could potentially actuate. What we actually find is that by increasing the number of threads updating a shared data structure that is protected by a lock, the performance actually decreases as the number of threads increase. The overall performance achieved is a very small fraction of the ideal speed, which can only be seen in single threaded programs.

This is a limitation which severely hampers the overall execution speed of multi-threaded programs. It is compounded with the fact that computers are being manufactured containing more and more cores to achieve the speed increase which consumers grew accustomed to while Moore's Law held. Processors are now heading in the direction of becoming many-core which is similar to a graphics card in approach. This problem is all the more important to solve, as future programs will become more and more reliant on multi-threading for performance.

1.2 Potential Solution

A potential solution to this problem is to take an approach similar to the approach taken by Databases. If a thread would update a shared data structure that is not protected by a lock, it will do so speculatively. If there is a conflict between one thread and another, instead of the data structure becoming corrupted, one of the threads will rollback its speculative changes allowing the other thread to commit its changes thereby making them viewable to every thread. This is known as Transactional Memory and was first proposed, to be handled completely in hardware, by Herlihy and Moss [Herlihy

and Moss, 1993].

This concept is a logical extension of Load Locked (LL) and Store Conditional (SC) instructions which were first proposed by Jensen, Hagensen, and Broughton for the S-1 AAP multiprocessor [Jensen et al., 1987] for Lawrence Livermore National Laboratory (LLNL). These instructions have also been implemented in architectures such as PowerPC and ARM architectures. The implementations differ between architectures in the limitations of the size of the memory block that is covered by the instructions and the retry policies used.

Currently only two manufacturers, IBM and Intel, have implemented Hardware Transactional Memory (HTM) and even this is in the last 5 years. This project will use the Intel version because it is cheaper to experiment with as they have incorporated this into their general purpose chips without errata as of the Broadwell series. IBM appear to have only included HTM with their high end Series-Z [Jacobi et al., 2012] processor range. There is however, another aspect to this research. While just using HTM can greatly increase the performance of data structures that are inherently parallelisable such as a Binary Search Tree, this may not be the case for linear data structures such as an Ordered Linked List. The Ordered Linked List also happens to be the pathological edge case for a Binary Search Tree.

There is also a method to use multiple transactions to span a list instead of just using a single transaction. This has the potential benefit of improving the performance of a linear data structure as standard transactional memory is able to do for a parallel data structure such as a Binary Search Tree. This is known as the Split Transaction and was first proposed by Lev and Massen [Lev and Maessen, 2008].

1.3 Chapter Guide

The Background chapter will cover the state of the art research has been done into this topic to date. It will also contain some information to help the reader better understand the rest of the thesis to follow. The Design Algorithms chapter will detail the approaches taken to solving the problem outlined and how this was implemented. Results will contain the output data from the program outlined in Design Algorithms. This will be in graph and table format. The Discussion will explain what the results mean in the context of the application. The Conclusion will describe the results in the context of the Industry and how they could affect the design of future multithread applications.

Chapter 2

Background

2.1 Ordered Linked List

An ordered linked list is discussed here because it is the pathological worst case of a binary search tree. The ordered linked list in this case, is a singly linked list. A standard linked list or an unordered list is essentially an unordered stack which even though it is similar to a linked list, it behaves differently due to the difference in insertion and removal algorithms for it. Ordered lists are much easier to search for a particular key value as the entire list does not necessarilly need to be checked. This will perform poorly when compared to an ordered linked list when the list size is increased.

2.1.1 Origin of the Ordered Linked List

As stated earlier the Ordered Linked List is the pathological worst case of the Binary Search Tree. It is also however, a valid, and well used, data Structure in its own right. A more standard Binary Search Tree would have a structure similar to the example in figure 1. In this Tree the root node would be the first node entered into the Tree which in this case is 10. Any value less than 10 would go to the left and any value

higher would go to the right. If a value was equal to 10 nothing would happen as no duplicates are allowed in this Tree. This comparison style will continue for each node until each value seeking to be entered into the Tree is in place, or the key is a duplicate and therefore no insertion has taken place.

To remove a node, a value is checked against the root node. If the root node is to be removed, the rightmost node of the left subtree of the root node would become the root node. In other words, the highest value less than the root node would become the root in this situation. If the value to be removed is less than the root value, the comparison moves to the node along the left branch of the root node and here the comparison is repeated. While if the value is higher than the root node, then the comparison moves to the node on the right branch of the root node and the process repeats. With a matching value the same process for finding a replacement node occurs. If there are no nodes on the left subtree of the node to be removed, the node is removed and the previous node points to any child it might have in its right subtree. Finally if there is no matching node in the Tree, the method will exit without changing the Tree as early as possible.

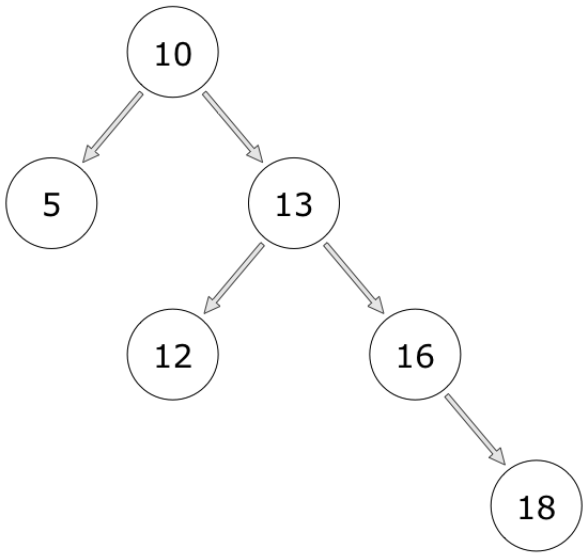

Figure 1: Binary Search Tree

For the sample Tree shown in Figure 1 to be the pathological worst case of a Binary Search Tree, the node with a key value 5 must be the root node. The remaining nodes shown must also enter the Tree in ascending order for this case to be realised. When this occurs, the Tree would look and behave like the Ordered Linked List shown in Figure 2. The difference between the two data structures at that point is that the Binary Search Tree could still resemble the Tree in Figure 1 after some favourable inserts and removals. It is very likely that this would occur given enough time. A very real concern exists where there is significant deterioration of the Tree leading to a data structure that resembles an ordered linked list. This is the case which should be prepared for when incorporating such a data structure into an application.

2.1.2 Data Structure Choice

For this reason, the Ordered Linked List, as can be seen in Figure 2, will be the data structure of choice for this research. The information which each node in the linked list will have is the Key and a pointer, Next, pointing to the subsequent node in memory

or NULL if there are no more nodes in the list.

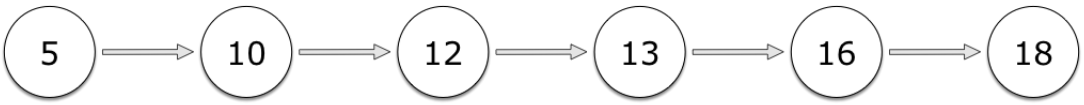

Figure 2: Ordered Linked List

To insert a node into the list, the steps used are very similar to those used in inserting a node into the pathological case of the Binary Search Tree. The key of the node to be inserted is checked against the key of the root node. If the value of the key is less than the root node's key then the new node becomes root and its next value points to the previous root node. If the value of the key is equal, no actions occur as the node already exists in the list so the insert terminates. While if the key value is greater than the root node's key, then the subsequent node undergoes the same comparisons. This is done exhaustively until the node is either entered in to the list or the insert is terminated as the value is already present in the list.

Removing a node from the list is the other main operation that can take place. The node with the key to be removed from the list is checked against the key value of the root node. If it is less than the key value of the root node, the node is not present in the list and so the operation terminates. If it is equal to the key value to be removed from the list, the next node in the list becomes the root node. If the value is greater than the key value of the root node, the next node in the list becomes the subject of these comparisons. However, there is a slight alteration in the process when a node, which is not the root node, is removed. The previous node in the comparison also needs to

be updated. This will point to the subsequent node to the node that is being removed. This process continues until the operation terminates due to the Key value not being present or the removal of a node takes place.

2.2 Transactional Memory

There are many implementations of transactional memory, both in hardware and in software. Software implementations have been available for some time and have been developed to a reasonably high standard. Hardware implementations however, are not as common. There are currently only two vendors who supply processors with Hardware Transactional Memory (HTM) support, IBM [Jacobi et al., 2012] and Intel. Sun microsystems were developing the Rock processor which reportedly had HTM support until the project was stopped when Oracle assumed control of Sun.

For the purposes of this research Intel's implementation of HTM was used. This decision was based primarily on the availability of HTM since IBM's version is only included in their high end Series-Z processors which are very expensive. Intel on the other hand have included HTM in all their processors as of the Broadwell series.

2.2.1 Origin of Transactional Memory

As stated in the Introduction, Transactional Memory is in essence an extension of the Load Locked (LL) and Store Conditional (SC) instructions [Jensen et al., 1987]. Unlike the implementations of LL and SC, it can examine to see if there are multiple memory locations and not just a single block of memory as in the LL and SC implementations. In order to do this, Transactional Memory has two sets, a Read Set and a Write Set. Any memory location that is read in by a Transaction is placed in the Read Set.

Similarly any memory location that a Transaction writes to, is placed in the Write Set. This model was first proposed by Herlihy and Moss [Herlihy and Moss, 1993]. It has become the basis for transactional memory research since its publication.

The main benefits of transactional memory, according to Herlihy and Moss, are to avoid locking issues such as Priority Inversion, Convoying and Deadlock. Priority Inversion occurs when a low priority task delays a high priority task. This is mitigated with transactional memory as each thread executes in parallel assuming there are enough cores for all the threads. Convoying occurs where a task has de-scheduled but that task has not released the lock. Using transactional memory this is avoided as no locks need to be taken. Since no locks are taken, no processes are blocked from running when a process is de-scheduled. The final issue that is avoided using transactional memory is Deadlock. This is avoided as no locks are taken so other processes running can not lock the same data set in a different order. This issue can also be mitigated with instructions such as a compare and swap instruction where the bus is also locked by the instruction.

In order for those attributes to be possible, a transaction would need to have a finite sequence of instructions which is atomic and serialisable. The atomic property is defined as the results from the instructions all committing as one or none. The sequence then restarts. The serialisable property is defined as the execution of a transaction will not interleave with other transactions. This only means that the finished transactions will appear to be serial in nature.

It is possible and probable that a transaction will update a memory location in use by another transaction. These transactions are then said to be in conflict. Conflict in this regard has defined states that are listed in figure 1. This table depicts the conflict state for only two transactions, A and B, as conflict of more transactions will be handled in

the same manner.

Case 1	Transaction A Writes to a memory location in Transaction B's Read Set.
Case 2	Transaction A Reads a memory location in Transaction B's Write Set.
Case 3	Transaction A Writes to a memory location in Transaction B's Write Set.

Table 1: List of Possible Transaction Conflicts

If a transaction does not encounter the conflict states shown above during its execution, it can finish or commit its results. A transaction will not be able to continue if one of the three states above are encountered. When this occurs, the transaction is said to have aborted. The transaction then has to be retried from the start and is referred to as a retry.

On a commit, a transactions changes must become visible to the rest of the transactions or threads as if this had happened in 1 cycle. This has to occur in 1 cycle for it to appear atomic to the rest of the threads. If a transaction is running in another thread and the commit happens in several cycles it would be almost impossible to have a consistent model. As a direct consequence of this, parallel logic is needed when a commit occurs to ensure all updated or added memory locations are seen in one block of changes. It also means that the changes can't be made to main memory as this would take too long. As a result all changes need to be published in the cache as this is much faster than accessing main memory. All changes are able to appear in a first level cache in one cycle. Using a snoopy cache coherence protocol, changes can be easily identified by other threads of execution.

The paper by Herlihy and Moss also proposed a validate instruction to report if a transaction had aborted. The emphasis is on the programmer, in their model, to test

if a transaction had aborted when it would be more progressive to have a section of code for handling the aborted transaction and the calling of this handler would be done in hardware. It should also be the case that an aborted transaction should not be running and the program counter of the processor should be pointing to the point at which the transaction had started. An aborted transaction should be shut down as soon as possible.

2.2.2 Intel's Implementations of Transactional Memory

Intel's HTM has eager conflict detection as a feature. As soon as any of the conflicts defined in table 1 occur, they are resolved. In all three cases, Transaction B will abort so it can start from the beginning of its code sequence while Transaction A will continue.

To monitor the Read and Write Sets, it is assumed that Intel have extended their MESI cache coherency protocol rather than adopt another cache coherency model such as Hammond's [Hammond et al., 2004]. MESI works on a cache line granularity level. It can also be assumed for simplicity that Transactional Memory is only a feature of the first level cache and is not present in the second or third level caches. This assumption will hold for Write Sets, but Read Sets do not appear to follow this behaviour in testing. However, for simplicity we will assume this principle holds.

There are four bits per cache line to represent MESI and, as we assume, some extra bits to represent Transactional Memory. Any conflict is resolved by hardware in the cache which makes it very fast. This is required in order to obtain performance increases over traditional lock based approaches. This is also where HTM has a major advantage over Software Transactional Memory (STM). STM takes much longer to handle a conflict than HTM and it takes STM a lot of processing to implement a similar conflict detection

system across different threads of execution.

To facilitate HTM, Intel has supplied programmers with six new instructions. These are XBEGIN, XEND, XACQUIRE, XRELEASE, XTEST and XABORT. XBEGIN and XEND are for beginning a transaction and committing a transactions changes respectively. XACQUIRE and XRELEASE are the instructions for Hardware Lock Elision which will be explained further in section 2.2.3. The final two instructions, XTEST and XABORT, have a different role. XTEST will check to see if a transaction is currently running. This is very useful as the code for both paths transactional and the fallback path can be combined together. This instruction allows the programmer to isolate code which they only want to run when in a transaction such as calling XEND if a transaction is running or releasing the lock otherwise. XABORT is an instruction to abort the currently running transaction. This allows the programmer to handle cases where they need the transaction to abort swiftly. Such a case can arise if the lock is taken on the fallback path, and a transaction needs to abort as another thread is updating the data structure and is doing so without using Transactional Memory.

```
unsigned int status = 0;                    //Holds the return status of xbegin();
while(true) {                               //Keep trying
    status = _xbegin();                     //Store the return value of xbegin();
    if(status == -1) {                      //check the return is -1

    /*UPDATE SHARED DATA STRUCTURE*/

        _xend();                            //End the transaction
        break;                              //Code executed. Get out of loop to
            continue program execution
    } else {
        //The code for handling aborted transactions will be here
    }
}
```

Figure 3: Basic C++ Transaction Code

```
RETRY:  or eax, 0xFFFFFFFFh      ;status = -1
        xbegin L0                ;Start the transaction

L0:     cmp eax, 0xFFFFFFFFh     ;Check eax is still equal to -1
        jne L1                   ;If eax != -1 jump to L1

        ;UPDATE SHARED DATA STRUCTURE

        xend                     ;Finish the transaction
        jmp L2                   ;Go to L2

L1:                              ;Code for handling aborted transactions
        jmp RETRY                ;Go back to start to retry the transaction

L2:                              ;Continue program execution
```

Figure 4: Assembly Generated from figure 3

2.2.3 Hardware Lock Elision

Hardware Lock Elision (HLE) is an attempt to use transactional memory with lock based programs. Intel have two new instructions XACQUIRE and XRELEASE to accomplish this. With HLE, the first time a thread attempts to acquire a lock it will automatically acquire the lock. It will however, be run as a transaction. If the transaction aborts for any reason it will actually attempt to acquire the lock. If another thread already has the lock, then the transaction will abort and the thread will have to wait until the lock becomes free. The advantage of this is that it is very easy to add onto the standard lock approach. If the code is written in assembly, two instructions need to be added to the lock code. The XACQUIRE instruction is placed just before the lock is acquired while the XRELEASE instruction is placed before the lock is released. If the code is in C or C++, the compiler will have already pre-implemented these instructions in functions, so it is just a matter of using the function that acquires the lock with the HLE version and replacing the code for releasing the lock with the HLE version as well.

It was designed to be easily used by programmers who had already implemented lock

approaches so they could get some of the benefits that transactional memory provides with very little effort. In a worst case scenario, HLE will have similar performance as the lock based approach. In other situations there is the caveat that the program could be more effectively parallelised. This would most likely be used when a lock based system is retrofitted to take advantage of some of the benefits of transactional memory.

2.2.4 Drawbacks of Intel's Hardware Transactional Memory

Intel's HTM is not without its drawbacks. It does require a lock based fallback path to be implemented. The manner in which it is implemented means that some events, such as page faults, can't be resolved within a transaction. It is also possible that the transaction could read or write to too many memory locations. While there are only three states where transactions are in conflict, there are many more global states that will cause aborts to occur. For this reason, Intel give no guarantee that a transaction will ever commit [Intel, 2013]. The main difficulty then is determining when, not if, should the transaction be abandoned for a particular operation such as inserting a node into an ordered linked list.

2.2.4.1 Lock Based Fallback Path

The lock based fallback path, as stated, is something that is certain to happen. When it does occur, for the lock to be effective, transactions can't update the shared data structure while a thread has the lock. This must happen even if a transaction is currently running. To ensure this will happen, we leverage the potential conflict states by placing the lock in the Read Set of the transaction as soon as it starts. To cover the scenario where the lock is updated by another thread after a transaction has begun

but before the lock is placed into the Read Set of the transaction, a comparison must take place.

There are drawbacks to this as well. For example, if the thread is de-scheduled off the processor, other transactions still can't complete until the lock is released. This is where time can be gained or lost in the execution of HTM. If a transaction is unlikely to complete with repeated tries, it should go to the fallback path as early as possible. However, if it is likely that the operation could succeed, it should be given every opportunity to do so as a transaction.

Reverting back to a lock is expensive. All other threads can not update the data structure while this is held, as stated before. Hence, it is a priority to minimise this state as it reduces the potential parallelisation of the operations. Due to the need to minimise the fallback paths execution time, the lock in place must be very efficient. We can also have threads wait until the lock is released before trying a transaction. Since transactions can not operate on the data structure while the lock is held, this should stop needless transaction aborts. Sample code of this algorithm can be seen in figure 5. Before a thread can start a transaction, it must first wait for the lock, named lock_var, to be reset to 0. Then inside a transaction, the lock variable is read into the read set of the transaction and the transaction is immediately aborted if it is set. If it is set at a later point in time while the transaction is still running this will cause a conflict and an abort will occur as defined by the conflict states in table 1

```
unsigned int status = 0;
while(true) {
    while(lock_var) {
        _mm_pause;
    }
    status = _xbegin();
    if(status == -1) {
        //Transactional Area
        if(_xtest() && lock_var) _xabort();
    } else {
        //Abort Handling
    }
}
```

Figure 5: Transaction Code Update

The factors mentioned are just some of the major considerations and drawbacks of this system. The Bakery algorithm would enforce fairness with regard to threads entering the fallback path. However, if we assume that we can minimise the number of locks taken as a proportion of the completed operations, then enforcing fairness, which would solve starvation and liveness, isn't a major concern. Efficiency and speed of execution is a priority. We would like to return to transactional operations as quickly as possible as it offers the best performance return. For this reason, a simple lock such as the Test and Set or the Test and Test and Set should be used. The Test and Test and Set lock reduces main memory accesses over the test and set lock by taking advantage of cache coherency protocols. This lock is therefore the lock that should be used due to its speed and efficiency of execution.

2.2.4.2 Limitations of Intel's Hardware Transactional Memory

Transactional memory, despite what it may seem, is not the solution to all of our problems when it comes to executing operations in parallel. There is an overhead in creating a transaction. It takes some time to set up a transaction. We also have the fallback path which is controlled by a lock meaning that in a worst case scenario, Intel's

implementation of transactional memory will degrade to a slightly lower performance than the lock implemented in the fallback path. This is a consequence of the overhead of creating the transactions and can not be avoided.

The data structure used is integral to the performance of a transaction. Serial data structure such as a linked list, will have far more contention than their tree based counterparts. Skip lists could also help alleviate some of the problems which the serial nature of linked lists pose. The main issue with serial data structures is the amount of potential conflict that can occur. For example, the average depth to find a node in a binary search tree containing one million nodes is 22. That is compared to 500,000 for a linked list of the same size. While it is possible that a binary search tree, can in its pathological worst case. degrade to a linked list, it is very unlikely.

The Write Set of a transaction is also limited by the cache line size of the first level cache. A typical cache line will be 64 bytes in size. With variables of 8 bytes in size (64 bits), 9 writes to the same cache line to which a memory address is mapped by the set associative cache could cause an abort. This limitation would increase the number of locks taken and unfortunately it is unavoidable. This type of abort will probably be rare but if the data structure is in a contiguous block of memory mapped to one cache line and the transaction has more than 8 variables in its write set every time, the transactional memory will have its worst case performance as outlined above.

Due to the necessity of a lock implementation in the fallback path, if a thread acquires the lock and is descheduled, no transactions can run during this period. This is mainly problematic when the number of threads running exceeds the number of available cores on the processor. However, this is a real concern as a scheduling quantum is typically around 2 milliseconds. The thread is also not guaranteed to run during the next quantum if it was de-scheduled. This is at the discretion of the scheduler of the

operating system. On Microsofts and most Linux/Unix operating systems, it is possible to give a thread a high priority to allow it the best chance of running. This option is not available on the Mac OSX operating system. Some long delays can therefore occur until the thread is re-scheduled so it can complete its operation and release the lock.

2.2.5 Current Research with Hardware Transactional Memory

Transactional memory needs to be compared to the methods it attempts to replace, which in this case is lock implementations. Pankratius and Adl Tabatabai [Pankratius and Adl-Tabatabai, 2011] compiled an investigation where they compared the development process between development teams to implement a software transactional memory approach against a lock based alternative. It was found that the team using transactional memory had a working solution quicker than the team that used locks. However, the time difference evened out as the transactional memory team spent the time saved into tuning the code that controlled transactional memory for best performance. The team using locks took the same time to get a working application as the transactional memory team took to implement and tune the code for maximum performance. From this we can conclude two things. Firstly, transactional memory simplifies the development process for a development team. To achieve maximum performance, the transactional memory approach code needs to be tuned to get the best performance which takes time.

The simplification of the code will make applications easier to maintain and update with new functionality. Additionally, it is also conceptually easier to manage which could reduce the number of bugs present in the application.

So far very little investigation has taken place with regard to Hardware Transactional

Memory. Romano and Diegues are at the forefront of this with two papers in the past two years on this subject [Diegues and Romano, 2015] [Diegues and Romano, 2014]. These papers focus on Intel's version of HTM rather than the IBM implementation [Jacobi et al., 2012]. Much of the current research involves Intel's implementation rather then IBM's. A likely cause of this is probably down to the cost of a series-Z cpu from IBM, while Intel have implemented transactional memory across their chipset range.

Diegues and Romano in their 2014 paper [Diegues and Romano, 2014], incorporated their transactional memory solution into a GCC compiler. This was a mistake as it is unlikely to have the optimum solution with this approach. Different algorithms and data structures behave very differently and as they pointed out themselves, "there is no one size fits all solution". It can be viewed that integrating the approach into a compiler is trying to make a one size fits all solution. However, they do get some reasonable performance increases from using transactional memory and their dynamic tuning achieves the same result as their best variant approach. There is also no comparison to current ubiquitous techniques such as a lock based approach, which would be the best comparison as the performance increase is relative to the performance of such an implementation, not relative to transactional memory running on a single thread. When multiple threads require a lock to operate on a data structure, typically there is a performance reduction with increasing numbers of running threads. Therefore transactional memory with 8 threads could be 4 times faster than transactional memory with 1 thread but could have a relative speed increase of 10 over the lock based approach with 8 threads. The follow up paper in 2015 [Diegues and Romano, 2015] is a more extensive paper with regard to the various simulations run. It however, has the same faults that are listed above.

Yoo [Yoo et al., 2013] investigated if Intel's hardware transactional memory could improve the performance of TCP/IP in 2013. This paper focusses on high performance computing workloads and the improvement that transactional memory can provide for these situations. They found that transactional memory could boost performance by about 40% under real world loads but fell to just over 30% under very high loads. This paper shows how versatile transactional memory is and that it can be adapted to meet most multithreaded systems where there is the potential to use multiple threads to increase speed of execution.

Another target for potential performance increases is in memory databases. Karnagel [Karnagel et al., 2014] investigated the use of transactional memory to increase the performance of B+ Trees and the Delta Storage Index which is implemented in the SAP HANA in memory database. This study used Intel's transactional memory implementation. Karnagel found that by using transactional memory, there was a performance increase but also a code simplification. The study found that algorithms implemented scaled better and were easier to verify on top of providing performance increases. To further strengthen the performance claims, Leis [Leis et al., 2014] compiled a study using transactional memory for updating data in an in memory database. Leis found that by using Intel's transactional memory, almost all of the database transactions could be achieved without a lock thereby vastly increasing performance. This allowed Leis to have concurrent database transactions commit with a very low overhead.

The possible use cases of transactional memory, some of which are mentioned above, is only matched by its potential to change the way programs are written. Herlihy, whose paper [Herlihy and Moss, 1993] is the cornerstone of this field, speaks of such a potential change in a more recent presentation [Herlihy, 2014]. Herlihy highlights some of the more recent novel uses of transactional memory in this talk. He also presented the

concept that transactional memory will not replace locks but rather they need to work together. This is supported by the fact that a transaction needs a lock based fallback path in the Intel implementation otherwise they can not guarantee that a transaction will ever commit.

2.3 Split Transactions

Split transaction as proposed by Lev and Maessen [Lev and Maessen, 2008] is splitting one transaction into smaller individual transactions. This could be done for a variety of reasons but reducing potential conflict is a major reason. When using split transactions on a data structure such as an ordered linked list, the reduction in conflicts that can be provided, offsets the overhead of setting up more transactions. The overhead of setting up a transaction is not insignificant and that is one of the limitations of transactional memory as it stands. The amount of conflict that can be observed needs to be very high for this method to be considered as a viable option to take.

Diegues [Diegues et al., 2014] has shown that in cases where the number of aborts is very high, transactional memory actually performs worse than a lock based alternative. This case is where split transactions are of value and as stated above, can be found in the ordered linked list data structure.

In order to implement split transactions, the point at which one split transaction ends needs to be saved so the next split can start from that saved point. There is an obvious problem with this. What happens when that point no longer exists in the list by the time the next split transaction is due to start? This is not an easy problem to solve and causes most of the issues to do with split transactions.

Another problem, which is not as obvious but just as important for the performance,

is how an abort of a split transaction is handled. This problem extends the previous problem stated. This means that the solution needs to be extended to take this into account. Practically, the best situation would be where each individual split transaction can be retried so long as the saved starting point of the split hasn't been updated. Otherwise it would need to start at the very beginning of the data structure which could potentially waste a lot of effort and resources of the processor. However, even if it is possible to create a checkpoint, if that checkpoint is updated then it has to go back to the very beginning.

Lev and Maessen were part of the team in Sun Microsystems developing the Rock processor which was rumoured to have an implementation of transactional memory. This lends credence to their statement where they state full hardware support for true closed and open nested transactions is unlikely to be possible. Since Intel have not implemented the split transaction concept in hardware, it means that this tracking of the checkpoints of the split transactions will need to be done in software. Split transactions was already going to have additional overhead time costs due to the extra transactions being created but there will also now be the overhead of tracking the checkpoints of the split transactions in software. While this is not ideal, the effect of the split transactions will have to be greater to offset this overhead.

Surprisingly virtually no attention has been paid to split transactions since their suggestion by Lev and Maessen. This lack of research into these could have been influenced by the cancellation of the Rock processor that Lev and Maessen were working on when this paper was released.

2.3.1 Limitations of Split Transactions

Split transactions have numerous limitations some of which have been outlined above. These limitations are quite large and as such there is quite a lot to overcome. The biggest issue however, is that hardware support for split transactions is unlikely to designed. Unfortunately true open and closed nesting of transactions would be difficult to support in hardware, as stated by Lev and Maessen [Lev and Maessen, 2008], which is the reason for this. It then has to be a software algorithm, as stated before, which makes it expensive to implement. It is likely that a solution would have its own data structure stored in main memory to keep track of each node that has been updated.

The performance of transactional memory will already have been compromised by the fact that there is an extra piece of software controlling the transactions. This is compounded by the problems of the sequence of events upon an abort with a transaction. If any abort causes the operation to go back to the start of the list, there is potentially a massive loss in performance over longer lists as the work needed to get to the same point in the list is likely to be a lot more when compared to shorter lists.

While performance is an issue, it is possible to effect the performance by changing the split length of the transaction. However, the only other way to directly effect the performance dynamically is to alter the number of retries. This gives a very interesting situation. While it is beneficial, what can be changed is limited. This also limits your options if performance is not adequate. It is also crucial when performing this task that the split length is not increased excessively as this will have a major adverse effect on the performance.

2.4 Alternative Solutions

There are numerous potential solutions to the multithreaded approach for updating a shared data structure. Normally such a data structure would be protected by a lock but there are also other options. Lockless approaches which create an ordered list of threads that want to update the data structure can ensure the starvation free property. An example of this approach would be the Bakery Algorithm [Lamport, 1974] which is outlined in section 2.4.2.2.

2.4.1 Lock Approach

There are many locks that could be chosen as the base lock approach to compare with transactional memory. However, it is beneficial to choose a lock that is both fast and efficient. This would exclude using any lockless algorithms such as the Mellor Crummy Scott (MCS) [Mellor-Crummey and Scott, 1991], Bakery or Black and White Bakery algorithms [Taubenfeld, 2004]. A lock that is both fast an efficient with its bus traffic is the Test and Test and Set lock.

The lock approach is the current norm and as such it is the main point of comparison for transactional memory to evaluate, and quantify, the performance increase it provides, or performance decrease in certain situations. It is important to note a Test and Test and Set lock is not a fair lock. It doesn't ensure threads are starvation free and it doesn't ensure liveness of threads. It is deadlock free however, which is the minimum requirement for a lock. This allows it to run optimally as it doesn't attempt to solve other issues such as liveness or starvation in software as algorithms such as the Bakery or MCS lock do.

Due to its speed, the same lock that is used as a comparison is also used in the fallback

path of the transactional memory. The Test and Test and Set lock was chosen here for the same reasoning, as it is the lock based comparison for the transactional memory. It is very fast and efficient with its bus traffic. Too much bus traffic will slow down the whole system so this is an important consideration.

2.4.2 Lockless Algorithms

These are algorithms that do not require a lock, hence the name, but instead rely on clever algorithms to provide the same functionality. Most lockless algorithms are superior for satisfying thread liveness and being starvation free than conventional lock approaches such as a Test and Set or Test and Test and Set lock. Two examples of such algorithms are Hazard Pointers [Michael, 2004] and the Bakery Algorithm. This section will give a brief overview of these algorithms and highlights some of their strengths and weaknesses.

2.4.2.1 Hazard Pointers

One of the most common problems faced when multiple threads can update a data structure in parallel is the ABA problem. Due to the nature of transactional memory this problem is avoided completely. However, algorithms that allow multiple threads to update a data structure without obtaining a lock, such as hazard pointers or Treiber stack [Treiber, 1986], must contend with this issue. An example of the ABA problem is as follows.

Thread T_1 attempts to pop node A from the list in figure 6. Before T_1 can complete its operation, it is preempted to allow T_2 to run.

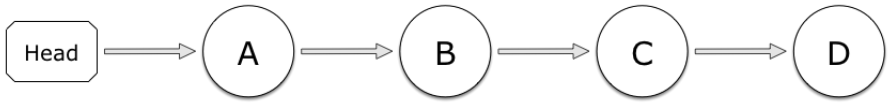

Figure 6: ABA Problem Diagram 1

T_2 will then pop node A. T_2 will push node X and node A onto the stack, as can be seen in figure 7. When node A is pushed back onto the stack, the memory location is reused.

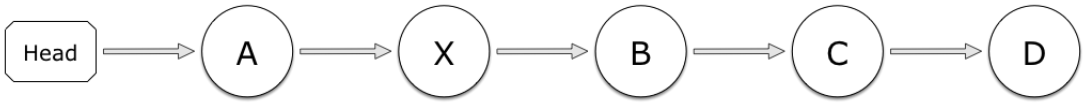

Figure 7: ABA Problem Diagram 2

T_2 operations complete and T_1 is allowed to run again. It is unaware of the changes that T_2 made to the stack. T_1 still thinks that node A points to node B. T_1 then finishes its operation which is popping node A off the list. With T_1 not knowing about the update to what node A is pointing to, T_1 moves the head to node B as can be seen in figure 8. This leaves the stack in an inconsistent state.

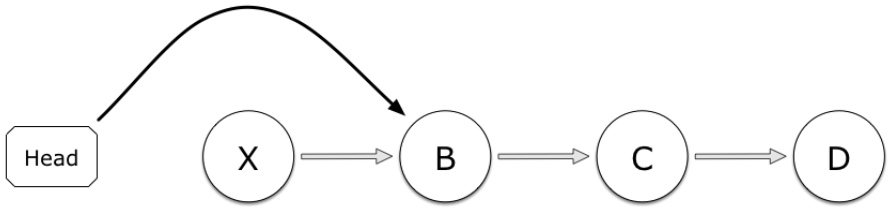

Figure 8: ABA Problem Diagram 3

One solution to this concurrent programming problem is to use Hazard Pointers [Michael, 2004]. Hazard Pointers solve this problem as each thread keeps track of all the memory locations that it is updating and stores the locations in the so-called Hazard Pointers.

This then allows for the implementation of a conflict detection systems to stop other threads updating memory locations another thread is using. If a thread is only reading a memory location, other threads will also be allowed to read that location but will not be able to update it. However, for a write, all other threads must wait until the thread which has that memory location in its hazard pointer, has updated it.

There are multiple implementations available in C and C++. This algorithm gives more benefits compared to a lock and is very similar to software transactional memory conceptually in with the problems it attempts to, and succeeds, solve. Each threads hazard pointers must be available to all other threads and the checking of these pointers can take time especially if a lot of memory locations are being updated.

2.4.2.2 **Bakery Algorithm**

In the 1970's Dijkstra's concurrent programming problem was a major issue [Dijkstra, 1965]. This problem is about ensuring the mutual exclusivity of a programs critical section where multiple threads can execute this section. This is the issue which conventional locks solve by only allowing one thread at a time to be in the critical section. One of the most famous solutions to this problem, and an alternative to transactional memory, is Lamport's Bakery algorithm[Lamport, 1974].

The Bakery algorithm is one of the first lockless solutions to Dijkstra's concurrent programming problem. It works by giving each thread a number that is one more than the maximum number assigned to a thread. Then each thread will in turn have their turn at executing the critical section in order. On exiting the critical section the number assigned to the thread is reset to 0. The main ingenuity about this algorithm is the way it deals with which thread should advance into the critical section when there are N threads running.

```
begin integer j;
  L1: choosing [i] := 1;
      number[i] := 1 + maximum (number[1], . . . , number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number [j], j) < (number[i],
                  i) then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

Figure 9: Lamport's Bakery Algorithm

Lamport's algorithm, shown above, requires read and writes to be in order for it to work. This memory fence needs to be placed anytime the choosing array is updated. If this does not occur it is possible for two threads to have the same max value which will cause the algorithm to break down.

By virtue that none of the processes can go through deadlock or live-lock, starvation free is also satisfied since there would be no other way in this algorithm for starvation to happen. These properties are crucial as most lock implementations, such as the Test and Test and Set lock, do not guarantee liveness which implies that starvation is possible. Deadlock free is a minimum requirement for all locks to have mutual exclusion on the owner of the lock.

Unfortunately the algorithm is not bounded and the maximum value assigned to a thread can keep going up until register overflow occurs. This problem was solved by Taubenfeld with the Black and White Bakery Algorithm [Taubenfeld, 2004]. Taubenfeld's algorithms achieves this by having two lists, a white list and a black list. While

the threads in the white list are in the critical section, the black list is filling up. Once all the threads in the white list have executed, the black list threads start to execute the critical section. While this happens the white list will start to fill up. The maximum values for the black and white lists are also independent. This has the effect of bounding the maximum values which can be assigned to the threads which in turn stops register overflow from occurring.

$choosing_i$:= `true` /* beginning of doorway */
$mycolor_i := color$
$number_i := 1 + \max(\{number_j \mid (1 \leq j \leq n) \wedge (mycolor_j = mycolor_i)\})$
$choosing_i$:= `false` /* end of doorway */
for $j = 1$ **to** n **do**
  await $choosing_j$ = `false`
  if $mycolor_j = mycolor_i$
  then await $(number_j = 0) \vee ([number_j, j] \geq [number_i, i]) \vee (mycolor_j \neq mycolor_i)$
  else await $(number_j = 0) \vee (mycolor_i \neq color) \vee (mycolor_j = mycolor_i)$ **fi**
od
critical section
if $mycolor_i$ = `black` **then** *color* := `white` **else** *color* := `black` **fi**
$number_i := 0$

Figure 10: Taubenfeld's Black and White Bakery Algorithm

The Bakery algorithm does have a major performance issue when the number of threads exceed the number that can run at once on a processor. The performance in these cases will be a fraction of the performance otherwise. It is also slower than other locks such as the Test and Set lock or the Test and Test and Set lock. Transactional memory will outperform the Bakery and Black and White bakery algorithms by a wide margin. This is down to the fact that the Bakery and Black and White Bakery algorithms do not perform as well as locks such as the Test and Test and Set locks which transactional memory outperforms. This was shown by Diegues and Romano [Diegues and Romano, 2014].

Chapter 3

Design

The main focus of this project was to discover the possible performance increase on ordered linked lists using transactional memory. Long linked lists will have a very large read set in comparison to a Binary Search Tree of the same size. The more memory locations in the read set, the more likely it is for an abort to occur. This is where split transactions can help. They will reduce the number of memory locations in a read set on average which should provide a performance increase over both standard lock implementations and transactional memory with no split transactions.

3.1 Baseline

The base comparison for all methods is that the lock based approach. The lock approach represents the current fastest way to update a shared data structure, therefore it will be used as the comparison for all other algorithms that are implemented and not the single threaded version of each algorithm.

```
volatile int lock_var = 0;
#define ACQUIRE()    {
                         do {
                             while(lock_var == 1) {
                                 _mm_pause();
                             }
                         } while(InterlockedCompareExchange(&lock_var, 1,
                             0) == 1);
                     }

#define RELEASE()    lock_var = 0
```

Figure 11: Test and Test and Set Lock

The lock shown above in figure 11 is the implementation of the Test and Test and Set lock that is used. The inner while loop limits the number of times that the InterlockedCompareExchange will run. Since the InterlockedCompareExchange is a Compare and Swap instruction, it will lock the bus and stop other threads from using the bus. This will delay and adversely affect the performance of other threads which is why it needs to be limited. The _mm_pause() is recommended by Intel when spinning on a variable that is in the cache.

3.2 Implementation

Hardware Transactional Memory is still in its infancy and as such there is limited information available on how to use it. Luckily GCC, Visual Studio and Clang have implemented functions as part of their C++ compiler allowing much easier usage of HTM. In Clang's and GCC's case, an extra command line argument is also needed for this to run. Even though only 6 new instructions were added to the instruction set, the implementation means that the code can be written in C (Compiler Dependant) or C++ rather than Assembly.

3.2.1 Transactional Memory

A very basic and in some ways incomplete example can be seen in figure 3. It can be noticed that the area of the code to handle a transaction abort is not present. This section and others were removed to simplify the example.

As can be seen below in figure 12 there are two more variables that need to be defined. Attempts holds the remaining number of tries that can be given to the transaction approach before reverting to a lock. Transaction is set to one for when a transaction should execute or 0 when the fallback path should execute.

```
unsigned int transaction = 1;
unsigned int status = 0;
unsigned int attempts = 8;
while(true) {
    while(lock_var) {
        _mm_pause;
    }
    if(transaction == 1) {
        status = _xbegin();
    } else {
        ACQUIRE_LOCK();
        status = _XBEGIN_STARTED;
    }
    if(status == _XBEGIN_STARTED) {
        if(_xtest() && lock_var) _xabort(0x01);

        /*UPDATE SHARED DATA STRUCTURE*/

        if(_xtest()) _xend();
        else RELEASE_LOCK();
        break;
    } else {
        if (attempts > 0) {
            attempts--;
        } else transaction = 0;
        while(lock_var) {
            _mm_pause;
        }
    }
}
```

Figure 12: Full Transaction Code

As can be seen in figure 12, the functions relating to the new instructions provided by Intel are named the same. Also within GCC they have defined _XBEGIN_STARTED

to be equal to -1 as this is the value that is returned in the register eax, the register all functions will put their return value in, if a transaction successfully starts.

There are some easy performance increases implemented in this code as well. A transaction has to be aborted if it starts and the lock is set. It therefore makes sense to not start a transaction if the lock is set. This can be seen with the first while loop inside the main while(true) loop. This loop is identical to the waiting loop used in the Test and Test and Set lock. This loop was also implemented at the end of the fallback path at the very bottom of the code. This will stop the transaction from going back to retry if the lock is set. These two changes to the logic, stop transactions from needlessly aborting because one thread has acquired the lock. This reduces the number of locks taken by limiting preventable aborts.

When using transactional memory it is very important to do all memory allocation outside of a transaction because malloc may be implemented with a lock which could cause the transaction to fail. Malloc is also very time consuming which could also increase the likelihood of an abort, even if malloc is implemented using an instruction such as a Compare and Swap instruction. Therefore to limit the number of calls to malloc, there needs to be a pool of unused nodes that a thread can use. When this is empty and only then, will a call to malloc be made. This will be done outside of the transaction in order to reduce the time in a transaction.

To reduce the complexity of this solution, each thread will have its own pool of allocated nodes available for use instead of calling malloc. This pool of nodes will fill when a thread removes a node from the list and adds that node to its thread local stack or "pool" of unused nodes. If an insert would occur but that node is already in the list, then this node will also be added to the stack of unused nodes. Each stack is thread local to remove shared data issues which could arise from a shared stack or "pool" of

unused nodes. This helps offset the time that malloc would take to run and hopefully the program would reach a point where malloc would no longer need to be called. These thread local stacks of nodes will only be freed when the thread is finished running to give the maximum potential increase from pooling unused nodes. This are no nodes present in the stack when the program begins execution. All nodes that are entered into each stack have either just come from the list, or the key value was already present in the list and therefore the node could not be added. These nodes need to be added outside the transactions to limit the time spent executing transactionally, which is not an issue as each stack is thread local.

3.2.2 Split Transactions

This code can then be extended to allow for split transactions to take place. Split transactions however, need some way of keeping track of updated nodes. An updated node is a node that has been added or removed from the list. One way of implementing such a system is to take advantage of the fact that the list is dynamically allocated. It is then possible to map the memory address that the node is at to an index in an array. A 64 bit architecture was in the machine that was used for testing. This means that all the memory addresses are also 64 bits in length. In a 64 bit architecture, the three least significant bits are 0, which allows the mapping to be done with the three least significant bits shifted out.

To map the address to an index in the array, the remainder operator in C++ (%) is used on the address with the length of the array. The remainder is that memory addresses index in the array. A check can therefore be done at the start of a transaction to ensure that the index corresponding to the memory location that the split transaction is starting from has not been changed. It is crucial that when a node is updated,

the index in the array corresponding to the memory address is incremented in the transaction. If it is updated outside the transaction, there is no guarantee that list will be consistent as two transactions could theoretically update the same node. It is also important to set each index in the array to 0 before starting. This is because wrap around could occur as some very large numbers could have been present in the memory locations now assigned to the array.

With this approach, the memory address of the position to start the next split transaction from can not be stored in a pointer. It has to be stored in a variable that is not dynamically assigned. The address needs to be stored in a 64 bit unsigned variable. Then it can be cast back into a pointer when needed. Just in case that memory location has been updated the variable can be shifted 3 bits to the right and the corresponding index can then be found. If the value at that index has changed, we need to start from the very beginning of the list again. If it has not been updated, the next split can be started from this location.

This logic can be extended to deal with the case where a split aborts. Using the same logic, it is possible to retry the same split so long as the index corresponding to the memory location of the start of the split has not been changed. This can increase the performance of split transactions as we are eliminating work that shouldn't need to be repeated unless the checkpoint or saved memory location has been updated. An example implementation of the algorithm can be seen in figure 13.

```
    int mode = TRANSACTION;                          //Transaction == 1
    unsigned int attempts = MAXATTEMPT;
    unsigned int status = 0;
    unsigned int split = 0;
    unsigned long long save = 0;
retry:
    Node* volatile pp = head;                  // head of the list
    Node* volatile p;
    if(mode == TRANSACTION) {
        unsigned long long = TAG(pp);
nextsplit:
        int backoff = 0;
        int cnt = 0;
        while (1) {
            status = _xbegin();
            if (status == _XBEGIN_STARTED) {
                if(split == 1) pp = (Node*) save;
                if  (lock_var)
                    _xabort(0xA0);
                if (myTag != TAG(pp)) {      //abort if tag corresponding
                    to saved position is different to saved value
                    split = 0;
                    _xabort(0xA1);
                }
                /* UPDATE HEAD HERE */
                do {
                    /* UPDATE SHARED DATA STRUCTURE HERE (BAR HEAD) */
                    pp = p;
                    if (++cnt >= split_length) {          //if the nodes
                        passed in list == split length
                        myTag = TAG(pp);                  //save the
                            number corresponding to the tag
                        save = (unsigned long long) pp;   //point at
                            which to start next transaction
                        _xend();
                        split = 1;                         //indicates
                            next transaction is following on from a
                            previous split
                        goto nextsplit;
                    }
                    p = p->next;                           // go to next
                        node in list
                } while (p && p -> data != newNode -> data);
                _xend();
                return 0;
            } else {    // here if transaction aborts

                if (--attempts <= 0) {
                    mode = LOCK;
                    goto retry;
                }

                if(split == 1) goto nextsplit;
            }
        }
```

Figure 13: Full Split Transaction Code

3.3 Dynamic Tuning of Split Transactions

There are two ways to influence the performance of split transactions on the fly. Firstly, the number of retries that a transaction can have can be altered and secondly, the split length of the split transaction can also be altered. Both of these methods will have different effects on the performance of the splits, but they share one characteristic. More retries will increase the performance where contention is low while it will decrease the performance where contention is high. Under high contention scenarios, it is usually more beneficial to take the lock sooner to limit the time wasted with aborted transactions.

Changing the split length has the exact same characteristics but the performance results will be more drastic in comparison. The longer the transaction, the more likely it is that there will be contention, especially as the number of threads rise. Both of these avenues for influencing the performance are therefore double edged swords. They could severely hamper performance by increasing the number of retries or split length too quickly, but conversely, being too slow to increase the values will lower performance in the split length case. With the number of retries increasing, it could have an adverse effect on the number of locks that need to be taken.

3.3.1 Design

The potential performance outcome is a huge consideration when choosing how to implement performance tuning algorithms. When changing the split length, there are a lot of advantages and this approach can be quite conservative when it comes to increasing the split length. When the split length is increased too far, it will severely hamper performance by causing too many aborts. However, the issue with smaller split lengths is a slightly smaller drop in performance, but it is more likely to complete.

An alternative approach to the problem that can be implemented in conjunction with the approach stated above is to be really aggressive when reducing the split length in the case of an abort. A very aggressive approach to this is in line with the slow increment of the split length that is mentioned above. This will lead overall to a very conservative approach to the tuning of the split transaction length.

This may appear simplistic, and it to some extent is, but the speed of execution of the tuning can be very fast. The reduction of the split length can be executed in 1 operation with a shift to divide the number by two. If this would reduce the split length below a minimum split length threshold, then the split length could be set to that threshold. Likewise, when increasing the split length, if the increase would bring the split length over the designated threshold, then it should set the split length to be that threshold. While these may not be the most radical or even inventive ideas, the concept seeks to limit the calculations necessary between operations on the list. This is of primary importance as a lot of time is already used by implementing split transactions.

3.3.2 Birthday Problem

The birthday problem is a method of discovering how likely it is for n people to have the same birthday. It can be adapted to find the probability that at least two threads will be in the exact same fixed section, or split, of a linked list. The standard birthday problem equation is defined as follows.

$$p(n) = 1 - \overline{p}(n) \tag{1}$$

Where

$$\overline{p}(n) = \frac{365!}{(365^{\mathrm{n}})((365-n)!)} \tag{2}$$

In equation 2, the references to 365 are the number of days in a year. Therefore if the references to 365 are replaced by the key range of the list divided by the split length, we will have a representation of how likely it is that two splits will be in the same split of the key range. We must also consider that, on average, there will be half of the key range in nodes present in the list. This can be stated because we will either attempt to insert or remove a node from the list with each attempt. Both insertion and removal will have the exact same probability of occurring. Therefore the key range will be divided by 2 to get the average number of nodes in the list. For this purpose n will be the number of threads instead of the number of people. Then it should be as follows.

$$\overline{p}_{\text{intersect}}(n) = \frac{\left(\frac{KeyRange}{2(SplitLength)}\right)!}{\left(\left(\frac{KeyRange}{2(SplitLength)}\right)^{\text{n}}\right)\left(\left(\left(\frac{KeyRange}{2(SplitLength)}\right) - n\right)!\right)} \tag{3}$$

This equation can give the probability that more than one thread will not be in a split assuming that the split length is fixed for the duration of the execution. To find the probability that more than one thread will be in the same split, subtract the result of the equation from 1. Even though computers use pseudo random number generators, the same result should not be gotten every run. The pseudo random number generator is seeded with the same number every time, but the interleaves of the threads is uncontrolled and entirely random. This should cause enough randomness in the system for this equation to hold over the long run.

The probability of a conflict can be derived using equation 3 which was derived from the birthday problem equation. On average, half the threads will operate on the list past the halfway point of the list. Half the threads will, on average, operate on the first

half of the list. Due to this it is possible to assume that on average, half the threads have the potential to operate on the list and potentially cause a conflict. However, when considering the probability that another thread could be in a split, all threads are considered as all threads could potentially abort. This model assumes only half the threads could cause the interference for aborts to occur. Therefore the equation to model the probability of a potential conflict is.

$$p_{\text{conflict}}(n) = p_{\text{intersect}}(n) \times \frac{n}{2\left(\frac{KeyRange}{2(SplitLength)}\right)} \tag{4}$$

It is important to note that this equation only deals with the average likelihood of a transaction aborting. There will probably be some factors during the execution of a transaction which could cause the figures obtained from equation 4, not to hold. One potential case where this will not hold is when 9 writes occur to the same cache line. This is due to Intel's cache coherency protocol which only has space for 8 memory locations in a cache line and thus the transaction is aborted.

Chapter 4

Results

In order for the tests carried out to be as relevant as possible, the variables in the system need to be kept to a minimum. The variables in the system need to be varied the same amount for all tests as well for the same reason. Some of the main variables in the system are as follows. The number of thread were varied for the execution of the program with 1, 2, 4, 8, 16 and 32 threads. For each of the six different values of threads, the program would be run for 10 seconds. One full execution on one key range with one constant split length would therefore take 60 seconds to complete. For example, with the static analysis that was done on a key range of 64, there are 7 different split lengths used ranging from 8 to 56. The program would have run for 7 minutes to get those results.

One problem running benchmark style tests like these, is to give the system time to stabilise. If this is not allowed, results from separate runs are likely to be vastly different as the list could still be filling which will artificially affect the results. On average only half of the insert operations will add a node to the list as half the time the key will already be present. The test would therefore have to run for a minimum where the total number of operation is 4 times that of the key range. Then to for this

artificial effect to be sufficiently reduced, the test would need to run for a time that allowed for the number of operations to be at least 10 times the key range. Through experiment the 65536 key range produced a minimum of roughly 5000 operations per second. This key range would have to run for 130 seconds as that is the time needed for the number of operations completed to be 10 times the key range. Alternatively to reduce this artificial performance boost, the list could be prefilled with all the odd numbered nodes in the key range.

The processor used is part of the Broadwell family from Intel. It is the first family of processors from Intel with HTM without errors in its implementation. The processor information is listed below in table 2.

Processor	Intel Xeon D-1540
Clock Frequency	2.00 GHz with 2.60GHz turbo boost
Memory	128GB
Number of Cores	8
Number of Threads	16
Architecture	64 bit
L1 Cache size	32KB
L2 Cache size	256KB
L3 Cache size	12,288KB (12MB)

Table 2: Processor Information

4.1 Static Analysis

In order to determine what the allowable split range should be, the number of transactional attempts an operation on a list can have are capped at 16. This allows for only one variable in the execution of the program, the split length. The split length is varied from 8 to 56 on a key range of 64 and 8 to 120 on key ranges of 4096 and 65536. These split lengths were chosen as the performance of larger splits, over 120, drops off considerably. This should give an indication as to the limits, both upper and lower, for the split length in the tuning algorithm. To make the graphs easier to read, each key range has the total information spread over at least two graphs.

The graphs are laid out with the number of threads on the x-axis and the operations per-second on the y-axis. Operations per-second is the performance metric used in these tests. It is measured when an insert or a remove successfully complete either transactionally or by obtaining the lock in the lock based fallback path. Each line on a graph refers to a different split length used during the execution with the legend every graph to illustrate which lines correspond to the various split lengths.

4.1.1 Key Range of 64

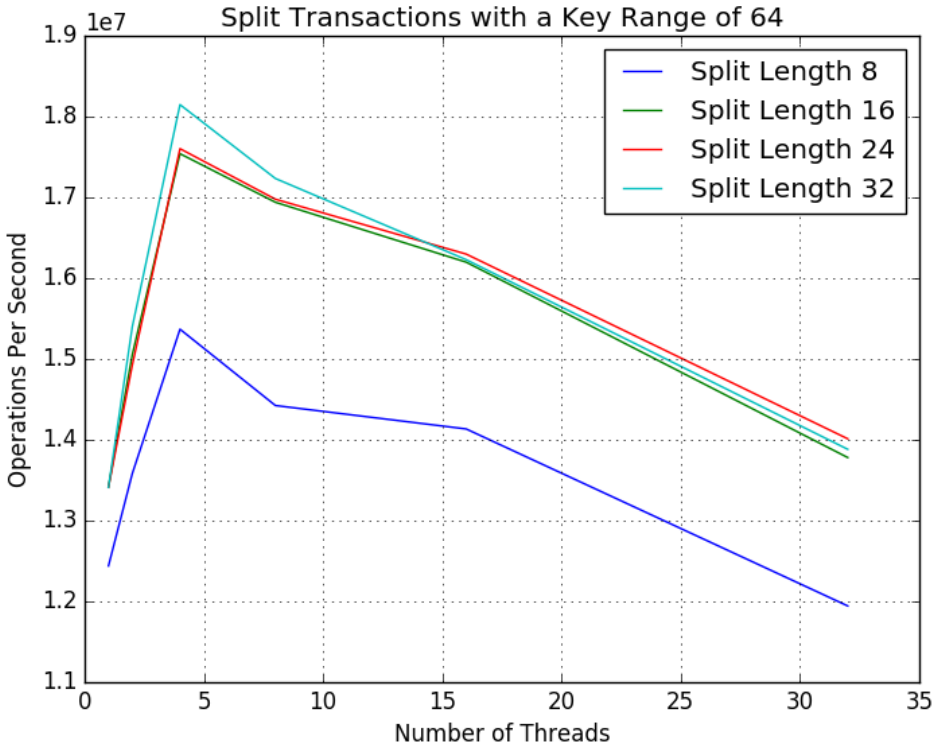

Figure 14: Static Split Length with a Key Range of 64 Part 1

As can be seen in figure 14, this graph covers the split range of 8 to 32 for a key range of 64. In this graph, the performance with a split length of 8 is well below the performance of all other split lengths. This indicates that a split length of 8 is not beneficial on such a small list. One of the reasons for this could be that on average with a key range of 64 there will be 32 nodes in the list. This will require 2 splits (16 nodes down the list) to reach the point of insertion or removal into the list on average. Whereas the rest of the split sizes all exceed the average number of nodes down the list a thread will go for an insertion or removal. With 4 threads the split length of 32 is the best performant which is interestingly the average length of the list. Beyond 4 threads the performance equalises for all split lengths.

There is a noticeable decrease in performance from 16 to 32 threads. This drop in

performance is caused by the context switching of the threads. As only 16 threads can run at any one time, every scheduling quantum there is the overhead of a context switch for each cpu. This phenomena will occur for all split lengths and key ranges. In fact would be very unusual and highly suspicious if there was no performance decrease where an application has the maximum number of threads a processor can run at a time and where it has double the maximum number of threads that can be run at a time. This performance drop could be exacerbated by a thread being preempted off the processor while holding the lock on the data structure as no transactions can update the data structure while the lock on the data structure is held.

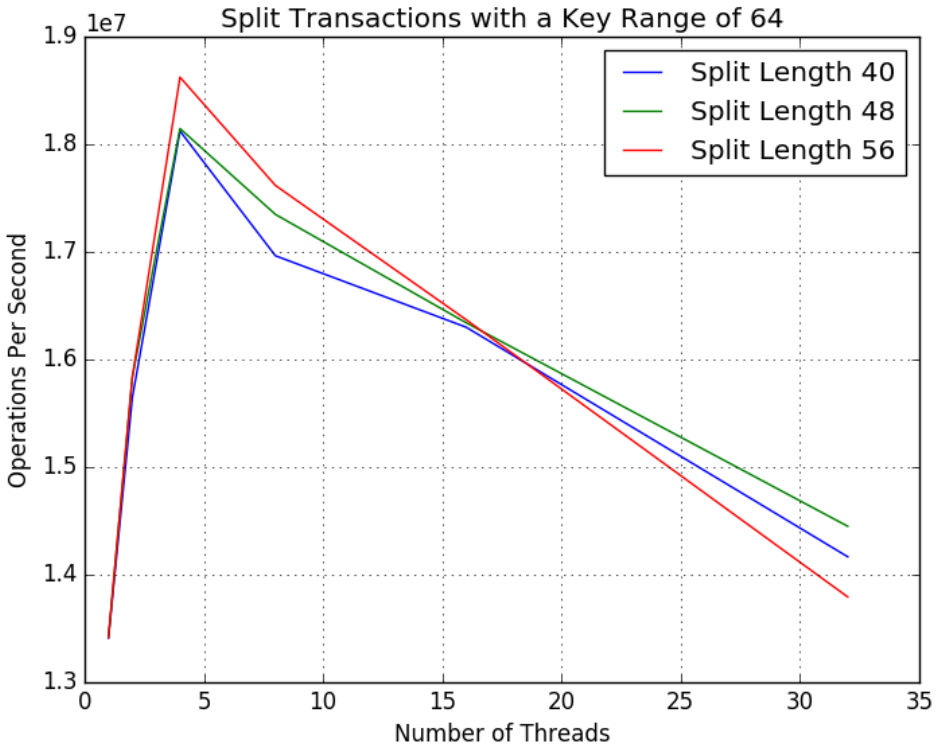

Figure 15: Static Split Length with a Key Range of 64 Part 2

A similar trend can be seen with regard to the performance of 4 threads in figure 16. The closer the split length comes to the key range, the better it seems to perform. This is due to the length of the list. Operations on the list can complete quite quickly

compared to those on an increased list size. When compared to graphs (figure 17 and 20) of similar split lengths for key ranges, 4096 and 65536 respectively, it is possible to confirm that the list length is key to performance outcome. This graph highlights the high cost of sharing data across too many threads and by extension cpu's.

4.1.2 Key Range of 4096

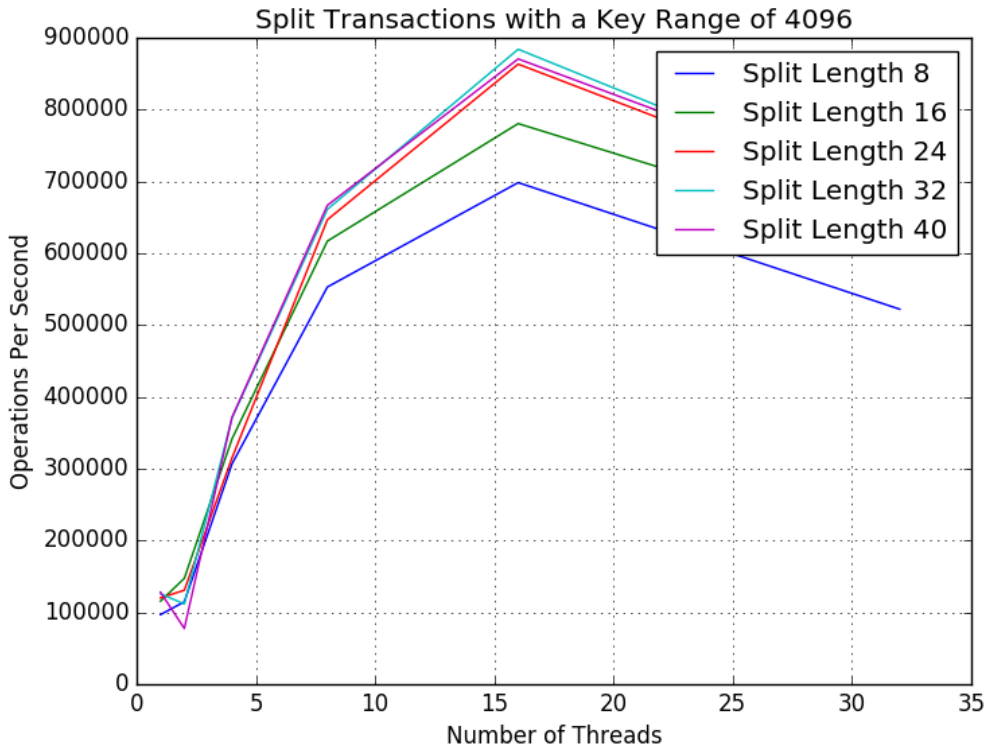

Figure 16: Static Split Length with a Key Range of 4096 Part 1

When the key range of the list is increased to 4096, there are some interesting differences with the results where the key range is 64. We can see that the performance is at a maximum at 16 threads and not 4 threads for the split lengths shown in figure 16. It is also interesting to note that the split lengths in figure 16 apart from split lengths of 8 and 16 decrease in speed from 1 to 2 threads. This is evidence of a large increase in the number of transaction aborts, and by extension, the number of locks.

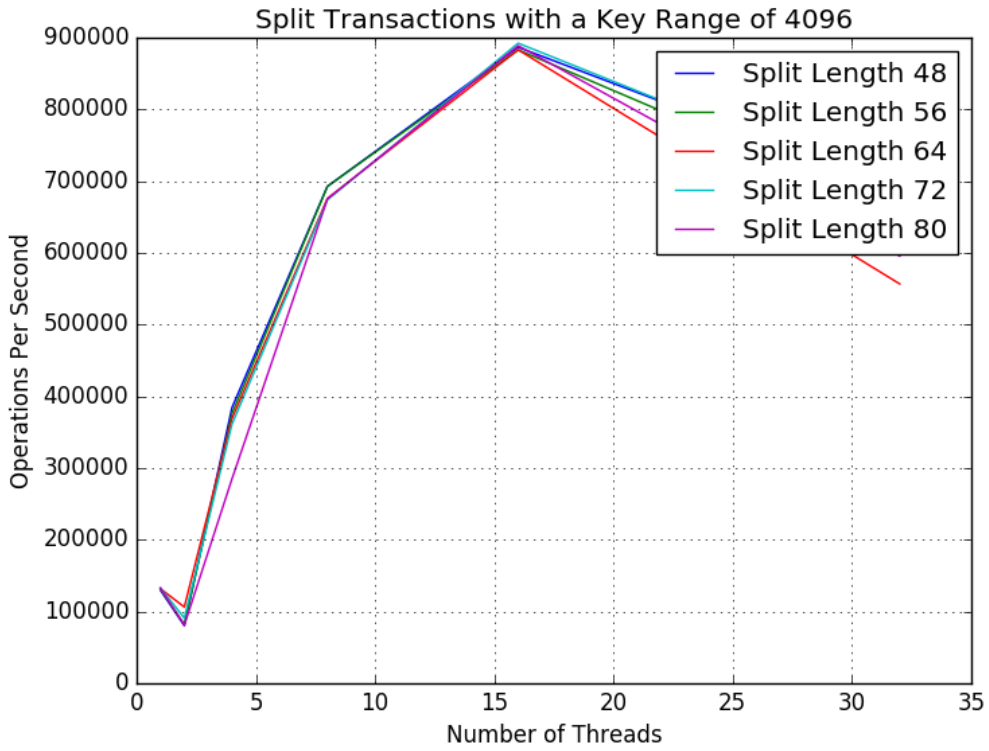

Figure 17: Static Split Length with a Key Range of 4096 Part 2

The trend that could be observed in figure 16 with regard to the performance of split lengths with 2 threads is even more striking in figure 17. We can see that there is now a large dip in performance between 1 and 2 threads which indicates that the issues seen in figure 16 have been exacerbated with an increase of the split length. However, we see a large improvement, if not a stabilisation, of performance with a greater number of threads. It appears that there is a large range of split lengths which will achieve almost peak performance for this key range.

Based on these results, the dynamic tuning of the split transactions will need to place the split length on the lower end of the spectrum when using two threads while allowing for an increase in the split length when there are more threads available for program execution.

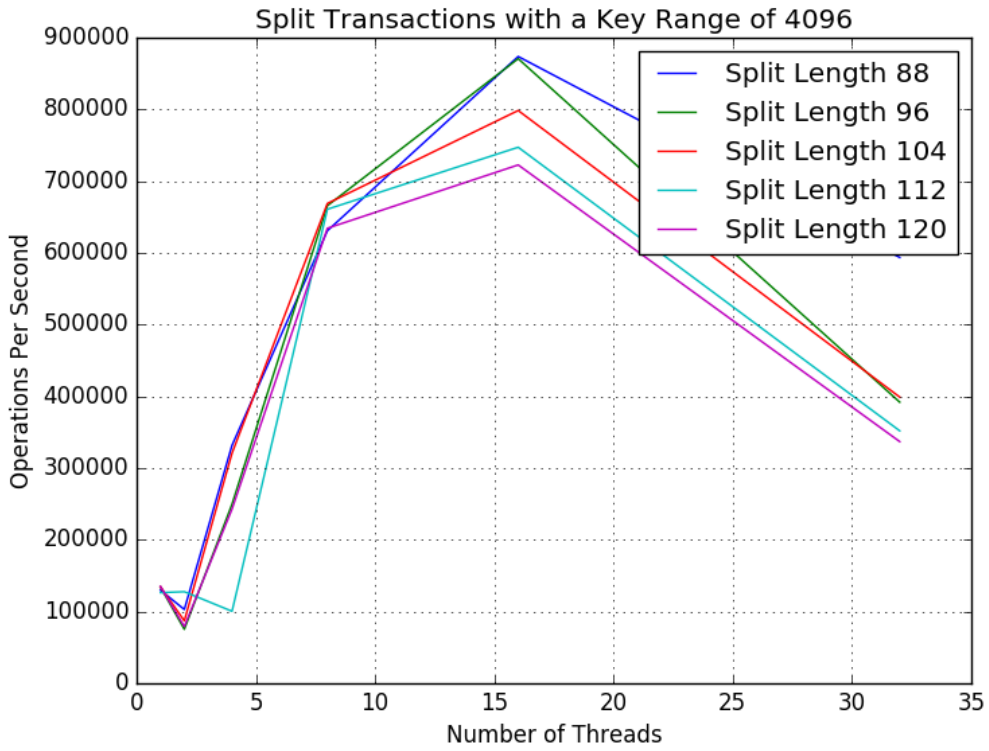

Figure 18: Static Split Length with a Key Range of 4096 Part 3

With a key range of 4096, we now see a decline in performance where the split lengths continue to increase as shown in figure 18. We can clearly see that with a split length above 88 with the key range of 4096, the performance drastically drops off after 8 threads. It should also be noted that the behaviour previously identified about the lack of performance with 2 threads is deteriorating. There is also what appears to be an anomaly with a split length of 112 where the performance drops off for 4 threads as well as for 2 threads. This could be an anomaly but if the split length is increased further, it is possible that it could be representative based on the rapid decline in performance that can be observed at 16 threads and above when compared to figures 16 and 17. This decline in performance is likely due the split lengths being too long which is causing more transaction aborts. An increase of aborts is very likely to cause a performance decrease.

Given these trends with the results, we know that above a split length of 88, there is a distinct lack of performance for a key range of 4096. It was also shown that the split length of 8 is needed to help overcome the performance drop off that which is apparent with 2 threads.

4.1.3 Key Range of 65536

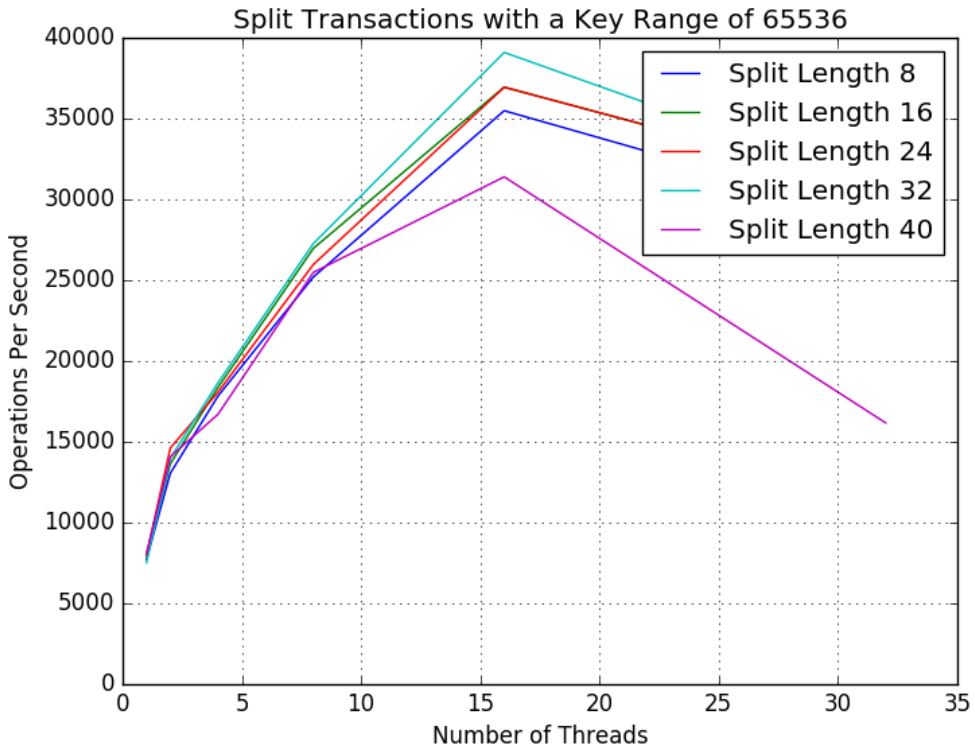

Figure 19: Static Split Length with a Key Range of 65536 Part 1

There are several very interesting pieces of information that can be obtained from figure 19. What we don't see is as interesting as what we do see. For example, there is no drop in performance with 2 threads compared to a single thread. We instead see a performance increase by a factor of 2 across the board with the operations per-second between 1 and 2 threads. This is most likely due to the fact that there is a greatly reduced chance of a conflict occurring with a key range of 64K (65536) as compared

to 4096. Particularly when the average number of splits needed to traverse the list is considered. With a key range of 4096 and split length of 32, on average half of the list will be filled. This means that on average there will be 64 splits that are 32 nodes long in the list. This compares with 1024 splits, with a length of 32 nodes which are in the list on average. In both cases the key range is halved to find the average number of nodes in the list.

The decrease in performance that can be observed in figure 19 as compared to figure 16, the first graphs for key ranges 64K and 4K respectively. It takes significantly longer for the performance to drop off in the shorter list length. This might seem a little counter intuitive as it may initially appear that having the same number of splits for a longer list as for a shorter list, should provide similar results relative to the single thread performance. However, this is not the case as is illustrated by the peak performance coming from a split length of 32 with a large decline to a split length of 40.

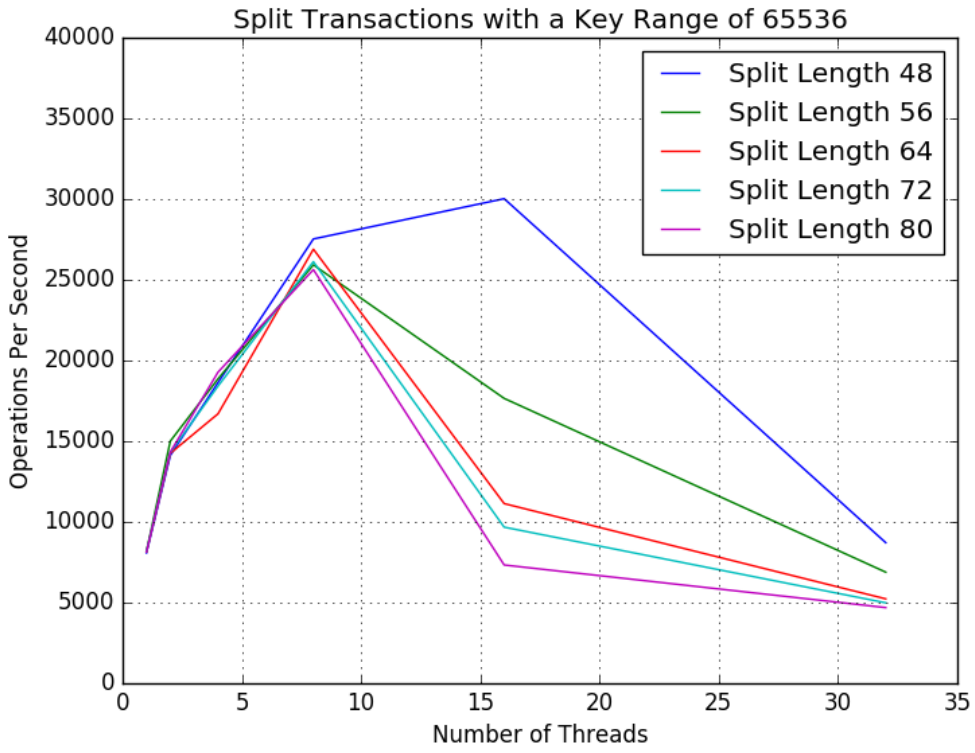

Figure 20: Static Split Length with a Key Range of 65536 Part 2

The trend seen in the first graph of the 65536 key range is followed on in figure 20. The split length of 40 is similar to the split length of 32. After that split length, the performance decreases severely. Despite the similarity for the split length of 40 to the split length of 32, the difference shows at 32 threads. Here the performance drops to the same level as the single thread performance. This is the start of a rapid crash of the performance. The rapid reduction in performance is at its most drastic where the split length is 80, is so severe that the performance of 16 threads drops to below the performance of a single thread. The performance up to 8 threads however, is very close to the performance shown in figure 19 which depicts the first 5 split lengths in the sequence used.

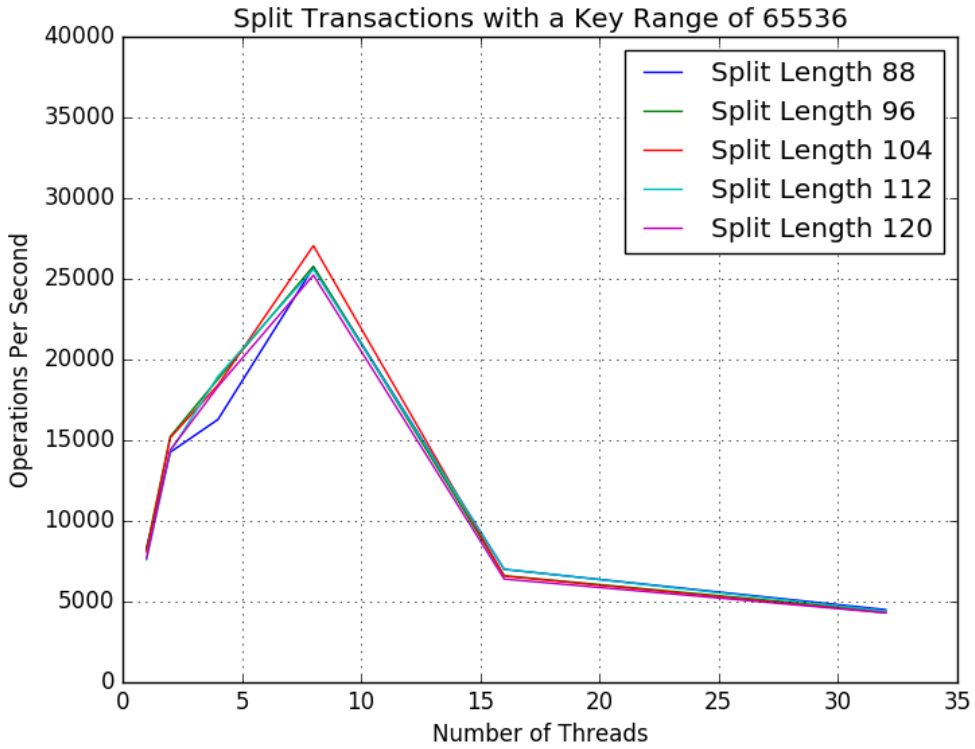

Figure 21: Static Split Length with a Key Range of 65536 Part 3

The final graph, figure 21, of the static analysis of the 65536 key range, continues the tendency shown in previous graphs. The performance, or lack thereof, stabilises with

a high split length value. This is consistent with results seen in the analysis of a key range of 64 and 4096. It can also be deduced that the split length range for this key range is 8 to 32. After 32 as the split length it was shown that the performance starts to drastically drop off.

4.2 Dynamic Tuning

Based on the results from the static analysis, bounds were applied to the split lengths. The upper bound was 32, while the lower bound was 8. This split length range provided the best performance across the static analysis for all three key ranges. The tuning algorithm used was the algorithms outlined in the design section.

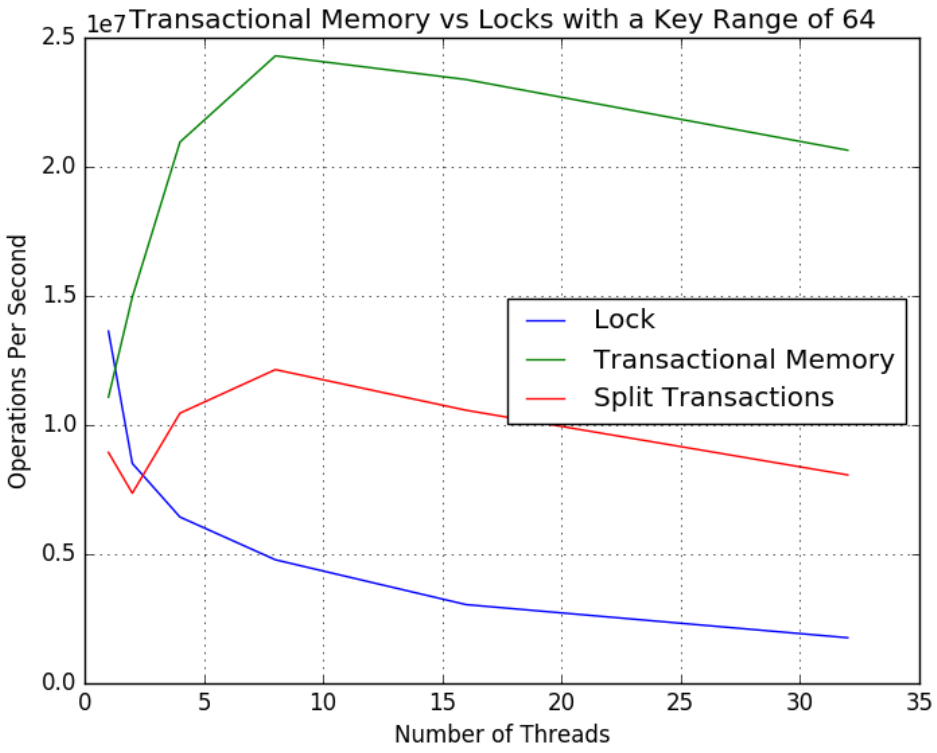

Figure 22: Key Range of 64

On short key ranges, such as a key range of 64, we have some interesting results. We

can clearly see in figure 22 that for short key ranges, transactional memory outperforms both the lock approach and split transactions. The performance difference of transactional memory against the transactional memory with split transactions is quite large. At its peak at 8 threads, transactional memory's performance 2x greater than split transactions. It is also 10x greater when compared to the lock approach for 16 and 32 threads. Split transactions have a performance increase of up to 4x over the lock based approach for 16 and 32 threads. Unfortunately we can see that updating the data structure with a single thread and a lock is faster than split transactions across the board and there is only a small performance boost provided by transactional memory with out split transactions.

The performance of transactional memory with no splits, is similar to the performance in the static analysis where the key range is 64 and the split length is 56 in figure 15. This is understandable as essentially the transactional memory approach is the same as having one split transaction go the entire length of the list.

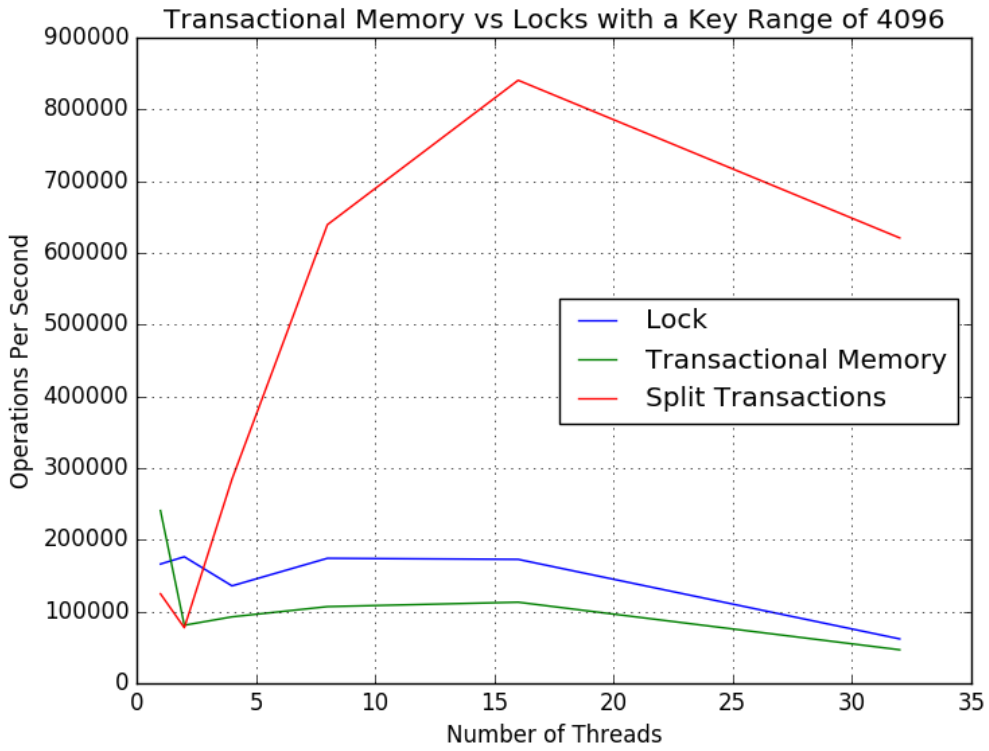

Figure 23: Key Range of 4096

The results obtained for a list with a key range of 4096 are much more promising. In figure 23, it is noticeable that split transactions far outperform both the lock approach and transactional memory that does not use split transactions. This is illustrated with a performance difference of over 6x at its peak compared to both the lock, and transactional memory.

In this example, transactional memory only performs as well as the lock in this case. Even though the chance of a conflict is greater over a shorter list, the transactions can execute much faster because of the length of the list. A second factor influencing transactional memory's poor performance, is the average workload required to get back to the same point in the list if an abort occurs. As the list is longer, after an abort it will, on average, take much longer to reach the point in the list at which the abort took place on a list with a key range of 64 or 4096. This extra processing has a large

negative impact on the performance.

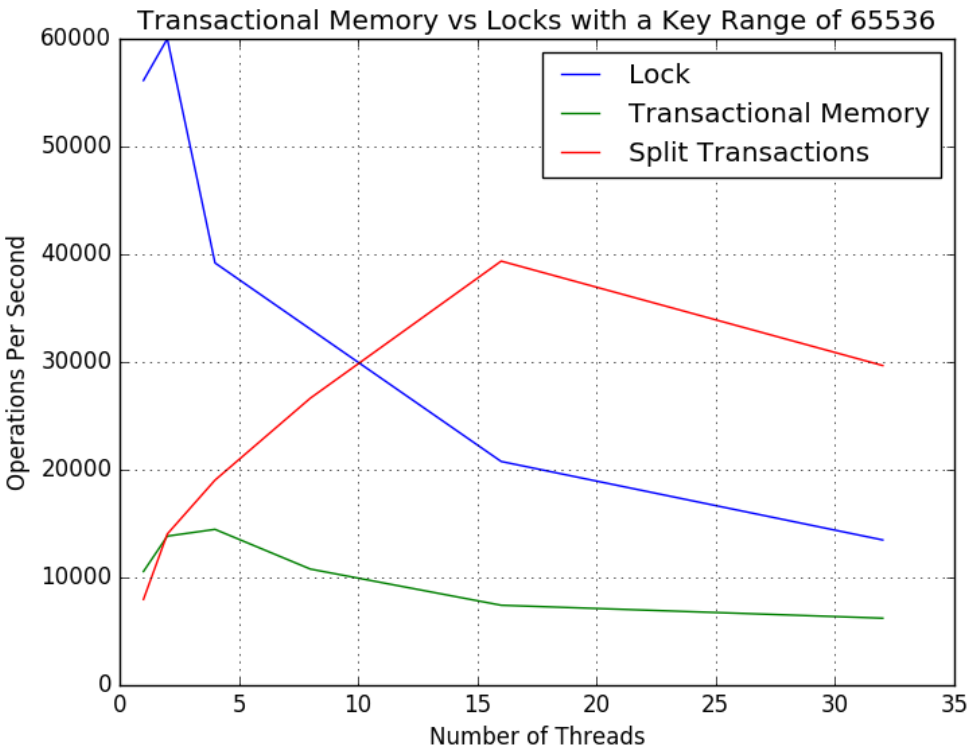

Figure 24: Key Range of 65536

With 64K as the key range, the results obtained are a little surprising. The best performant approach below 10 threads is the lock. The lock approach is in decline from two threads onwards. However, the lock approach is over 5.5x faster over a single thread when compared to both transactional memory and split transactions. However, split transactions show a marked performance increase with an increase in the number of threads. Transactional memory without splits transactions performed very poorly although with such a large key range it was not expected to perform well due to the very high probability of contention.

4.3 Conflict Prediction

Using the equations derived from the Birthday Problem equation, equations 3 and 4 which can be found in the Chapter 3 Design, it is possible to find the probability that two transactions will be searching through the same split in a list. The parameters needed for these equations are shown in table 3. It is crucial to note that in these results the thread value does not go above 16 even though all other tests reach 32 threads. The processor used can only run 16 threads at once, as shown in the processor information in table 2. Therefore only 16 threads will be able to interfere with each other at any point in time. This is why the calculations only go as far as 16 threads.

Key Range	Split Length	Average Length of LIst	Number of Splits
4096	32	2048	64

Table 3: Parameters for Conflict Detection equations derived from the Birthday Problem equations

In table 4, the probability that at least 2 threads will be present in any split at a point in time. This value is in the Probability in Same Split column. This value is then multiplied by a factor based on the number of threads and the likelihood of a thread operating in a split. This factor can be seen in equation 4. The result from this calculation is in the Probability of Contention column. The Probability of Contention is how likely a thread is to interfere with at least one other thread because at least one of those threads has operated in a split region that at least one other thread is in.

Number of Threads	Probability in Same Split	Probability of Contention
1	0	0
2	0.0156250000000002	0.000244140625000003
4	0.0910873413085939	0.00284647941589356
8	0.365972102659725	0.0228732564162328
16	0.87098847007459	0.108873558759324

Table 4: Results from the modified Birthday Problem equation, equation 3, and Conflict Detection equation, equation 4

This can be converted to the likelihood that a transaction will be a success by subtracting the value in the Probability of Contention column in table 4, from 1. Multiplying this by 100 gives the percentage of this result, and that is the value in the Calculated Success % column in table 5.

Number of Threads	Calculated Success %
1	100 %
2	99.9755859375 %
4	99.7153520584106 %
8	97.7126743583767 %
16	89.1126441240676 %

Table 5: Calculated percentage of successful Split Transactions

The calculated success percentage needs to be compared to results obtained from running Split Transactions with a key range of 4096 and a split length of 32 to test the accuracy of the conflict detection. For this test the split transaction algorithm needs to

be updated. Instead of potentially restarting a transaction from a saved point we need to restart from the beginning of the list. Otherwise the probabilities of the threads running split transactions will be very hard to model.

Number of Threads	Observed Success %
1	99.99 %
2	97.41 %
4	94.67 %
8	93.34 %
16	92.13 %

Table 6: Observed Success %

There is a difference between the calculated results and the observed results. However, for a method that is not very complicated, this is not a large difference. This difference is small enough to come under to be mostly explained by experimental error. As we can see the difference in the success percentage between observed and calculated results is out by, at most, 4 percentage points. Given the likelihood that some experimental error is involved, the predicted results are very close to the actual results. Given more time it may be possible to produce more accurate predictions.

Chapter 5

Discussion

There have been some positive results for split transactions in the results presented. The dynamic tuner performed very well with a key range of 4096 and again, to a lesser degree, when the number of threads got above 8 on a key range of 65536. Split transactions were up to 6 times faster than both locks and transactional memory on this mid sized list.

While this result was not matched on the longest list used in the comparison, key range of 64K, it did outperform both transactional memory and the Test and Test and Set lock when the number of threads exceeded 8. At 32 threads, split transaction were over 4 times faster than transactional memory and over twice as fast as the test and test and set lock. However, both transactional approaches failed to reach the performance of a single thread with a lock. A possible reason for this lack of performance could be the number of attempts that were allowed before a lock was acquired. In the tests, to reduce the number of variables the number of attempts was set to 16. When it is considered how likely a transaction is to abort is these circumstances and how much time is wasted, on average, getting back to the point in the list where the abort occurred, the time is not insignificant.

The key range of 64 was very good for transactional memory while split transactions and the lock lagged behind in terms of performance. The very poor showing of split transactions on this key range is explainable. The overhead of setting up transactions on a list that is on average going to be 32 nodes long is too big to overcome. This is shown by how successful transactional memory with no splits is on this key range.

As the key range increases, it would make sense for split transactions to come into their own. This however, is not the case. After the 4096 key range where split transactions performed very well it then could not even achieve the single thread performance of the lock. The small caveat of this is that it did finish as the highest performing approach once the thread count went above 8. Its speed also kept increasing to 16 threads where it would be expected to have its maximum speed. Therefore there is there is some optimism that a different algorithm for tuning the transactions could cause this performance to rise even further. It should also be noted that transactional memory without the use of split transactions was the slowest in the 64K key range and that split transactions provided considerable performance gains compared to it.

Using equation 4 derived from the birthday problem equation, some interesting results were obtained. The results were very similar to the inverse of the percentage of locks taken on the 4K list. There were some differences but this is probably down to experimental error and insufficient runs for the probability to fully stabilise. If the test was to run for longer this error would lessen but the experimental error would still be present in this circumstance. This shows that some work could be done to find optimum split lengths with statistics and then applied to the implementation. The investigation mentioned would be part of future work on this subject matter. The predicted results are close enough to the observed results that this should be possible.

Chapter 6

Conclusion

Research into Split Transactions has been sparse at best. With the results in this paper, Split Transactions prove worthy of a greater consideration than they currently receive. The performance increase that can be observed on a list with a key range of 4096 is substantial. The 9x performance increase that can be observed with the key range of 4096 over a current method, the Test and Test and Set lock, is remarkable. With more investigation this could even be larger.

It is questionable whether a list with a key range of 65536 is too big. However, there is evidence of a performance increase with Split Transactions on this key Range. The number of retries granted were static, but 16 could have been too many for a key range of this size. Therefore further investigation would need to be accomplished with static split lengths and a variance of the number of retries to investigate this argument.

Split Transactions should not be used unless necessary. From the results it is possible to conclude that a key range of 64 is not large enough for Split Transactions to be warranted. Transactional memory without Split Transactions delivers admirable performance gains over a lock based alternative. Therefore Split Transactions should only be a secondary method implemented if Transactional Memory does not provide

adequate results.

The surprising accuracy of the prediction of the success probability of Split Transactions could account for interesting future research. Based on the results, it appears to be possible to use statistical analysis to optimise the split length of a Split Transaction. This approach could also be applied to the dynamic tuner to potentially enhance the performance of the tuning of the split length.

Bibliography

[Diegues and Romano, 2014] Diegues, N. and Romano, P. (2014). Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 209–219. USENIX Association.

[Diegues and Romano, 2015] Diegues, N. and Romano, P. (2015). Self-tuning intel restricted transactional memory. *Parallel Computing*, 50:25 – 52.

[Diegues et al., 2014] Diegues, N., Romano, P., and Rodrigues, L. (2014). Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 3–14, New York, NY, USA. ACM.

[Dijkstra, 1965] Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–.

[Hammond et al., 2004] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. (2004). Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News*, 32(2):102–.

[Herlihy, 2014] Herlihy, M. (2014). Fun with hardware transactional memory. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 575–575, New York, NY, USA. ACM.

[Herlihy and Moss, 1993] Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300.

[Intel, 2013] Intel (2013).

[Jacobi et al., 2012] Jacobi, C., Slegel, T., and Greiner, D. (2012). Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 25–36, Washington, DC, USA. IEEE Computer Society.

[Jensen et al., 1987] Jensen, E. H., Hagensen, G. W., and Broughton, J. M. (1987). A new approach to exclusive data access in shared memory multiprocessors. Technical report, Technical Report UCRL-97663, Lawrence Livermore National Laboratory.

[Karnagel et al., 2014] Karnagel, T., Dementiev, R., Rajwar, R., Lai, K., Legler, T., Schlegel, B., and Lehner, W. (2014). Improving in-memory database index performance with intel 0x00ae; transactional synchronization extensions. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 476–487.

[Lamport, 1974] Lamport, L. (1974). A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455.

[Leis et al., 2014] Leis, V., Kemper, A., and Neumann, T. (2014). Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering*, pages 580–591.

[Lev and Maessen, 2008] Lev, Y. and Maessen, J.-W. (2008). Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory.

In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 197–206, New York, NY, USA. ACM.

[Mellor-Crummey and Scott, 1991] Mellor-Crummey, J. M. and Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65.

[Michael, 2004] Michael, M. M. (2004). Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504.

[Moore, 2006] Moore, G. E. (2006). Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Newsletter*, 20(3):33 – 35.

[Pankratius and Adl-Tabatabai, 2011] Pankratius, V. and Adl-Tabatabai, A.-R. (2011). A study of transactional memory vs. locks in practice. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 43–52, New York, NY, USA. ACM.

[Taubenfeld, 2004] Taubenfeld, G. (2004). *Distributed Computing: 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004. Proceedings*, chapter The Black-White Bakery Algorithm and Related Bounded-Space, Adaptive, Local-Spinning and FIFO Algorithms, pages 56–70. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Treiber, 1986] Treiber, R. K. (1986). *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center.

[Yoo et al., 2013] Yoo, R. M., Hughes, C. J., Lai, K., and Rajwar, R. (2013). Performance evaluation of intel® transactional synchronization extensions for high-

performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA. ACM.

Appendices

Appendix A

Code

A.1 helper.h

```
#pragma once

//
// helper.h
//
// Copyright (C) 2011 - 2015 jones@scss.tcd.ie
//
// This program is free software; you can redistribute it and/or modify it
    under
// the terms of the GNU General Public License as published by the Free
    Software Foundation;
// either version 2 of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY;
// without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
    PARTICULAR PURPOSE.
// See the GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software Foundation
    Inc.,
// 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.
//

#include "stdafx.h"          // pre-compiled headers
#include <iomanip>           // {joj 27/5/14}
#include <locale>            // {joj 7/6/14}

#ifdef WIN32
#include <intrin.h>          // intrinsics
#elif \_\_linux\_\_
#include <unistd.h>          // usleep
#include <cpuid.h>           // cpuid
#include <string.h>          // strcpy
#include <pthread.h>         // pthread_create
```

```
#include <x86intrin.h>      // need to specify gcc flags -mrtm -mrdrnd
#include <sys/mman.h>       // mmap, munmap {joj 23/5/14}
#include <limits.h>         // {joj 17/11/14}
#endif

#define AMALLOC(sz, align)  _aligned_malloc((sz + align-1) / align * align
    , align)
#define AFREE(p)            _aligned_free(p)

#ifdef WIN32

#define CPUID(cd, v) __cpuid((int*) &cd, v);
#define CPUIDEX(cd, v0, v1) __cpuidex((int*) &cd, v0, v1)

#define THREADH HANDLE

#define WORKERF DWORD (WINAPI *worker) (void*)
#define WORKER DWORD WINAPI

#define ALIGN(n) __declspec(align(n))

#define TLSINDEX DWORD
#define TLSALLOC(key) key = TlsAlloc()
#define TLSSETVALUE(tlsIndex, v) TlsSetValue(tlsIndex, v)
#define TLSGETVALUE(tlsIndex) (int) TlsGetValue(tlsIndex)

#define thread_local __declspec(thread)

#elif __linux__

#define BYTE    unsigned char
#define UINT    unsigned int
#define INT64   long long
#define UINT64  unsigned long long
#define LONG64  signed long long
#define PVOID   void*
#define MAXINT  INT_MAX
#define MAXUINT UINT_MAX

#define MAXUINT64   ((UINT64)~((UINT64)0))
#define MAXINT64    ((INT64)(MAXUINT64 >> 1))
#define MININT64    ((INT64)~MAXINT64)

#define CPUID(cd, v) __cpuid(v, cd.eax, cd.ebx, cd.ecx, cd.edx);
#define CPUIDEX(cd, v0, v1) __cpuid_count(v0, v1, cd.eax, cd.ebx, cd.ecx,
    cd.edx)

#define THREADH pthread_t
#define GetCurrentProcessorNumber() sched_getcpu()

#define WORKER void*
#define WORKERF void* (*worker) (void*)

#define ALIGN(n) __attribute__ ((aligned (n)))
#define _aligned_malloc(sz, align)  aligned_alloc(align, ((sz)+(align)-1)
    /(align)*(align))
#define _aligned_free(p) free(p)
#define _alloca alloca

#define strcpy_s(dst, sz, src) strcpy(dst, src)
#define _strtoi64(str, end, base)  strtoll(str, end, base)
#define _strtoui64(str, end, base)  strtoull(str, end, base)
```

```
#define InterlockedIncrement(addr)
    __sync_fetch_and_add(addr, 1)
#define InterlockedIncrement64(addr)
    __sync_fetch_and_add(addr, 1)
#define InterlockedExchange(addr, v)
    __sync_lock_test_and_set(addr, v)
#define InterlockedExchangePointer(addr, v)
    __sync_lock_test_and_set(addr, v)
#define InterlockedExchangeAdd(addr, v)
    __sync_fetch_and_add(addr, v)
#define InterlockedExchangeAdd64(addr, v)
    __sync_fetch_and_add(addr, v)
#define InterlockedCompareExchange(addr, newv, oldv)
    __sync_val_compare_and_swap(addr, oldv, newv)
#define InterlockedCompareExchange64(addr, newv, oldv)
    __sync_val_compare_and_swap(addr, oldv, newv)
#define InterlockedCompareExchangePointer(addr, newv, oldv)
    __sync_val_compare_and_swap(addr, oldv, newv)
#define _InterlockedExchange_HLEAcquire(addr, val)
    __atomic_exchange_n(addr, val, __ATOMIC_ACQUIRE | __ATOMIC_HLE_ACQUIRE
    )
#define _InterlockedExchangeAdd64_HLEAcquire(addr, val)
    __atomic_exchange_n(addr, val, __ATOMIC_ACQUIRE | __ATOMIC_HLE_ACQUIRE
    )
#define _Store_HLERelease(addr, v)
    __atomic_store_n(addr, v, __ATOMIC_RELEASE | __ATOMIC_HLE_RELEASE)
#define _Store64_HLERelease(addr, v)
    __atomic_store_n(addr, v, __ATOMIC_RELEASE | __ATOMIC_HLE_RELEASE)

#define _mm_pause() __builtin_ia32_pause()
#define _mm_mfence() __builtin_ia32_mfence()

#define TLSINDEX pthread_key_t
#define TLSALLOC(key) pthread_key_create(&key, NULL)
#define TLSSETVALUE(key, v) pthread_setspecific(key, v)
#define TLSGETVALUE(key) (size_t) pthread_getspecific(key)

#define thread_local __thread                                    // {
    joj 26/10/12}

#define Sleep(ms) usleep((ms)*1000)

#endif

extern UINT ncpu;                                                // #
    logical CPUs {joj 25/7/14}

extern void getDateAndTime(char*, int, time_t = 0);              //
    getDateAndTime {joj 18/7/14}
extern char* getHostName();                                      // get
     host name
extern char* getOSName();                                        // get
     OS name
extern int getNumberOfCPUs();                                    // get
     number of CPUs
extern UINT64 getPhysicalMemSz();                                // get
     RAM sz in bytes
extern int is64bitExe();                                         //
    return 1 if 64 bit .exe
extern size_t getMemUse();                                       // get
     working set size {joj 10/5/14}
extern size_t getVMUse();                                        // get
     page file usage {joj 10/5/14}
```

```
extern UINT64 getWallClockMS();                                    // get
    wall clock in milliseconds from some epoch
extern void createThread(THREADH*, WORKERF, void*);                //
extern void runThreadOnCPU(UINT);                                  // run
    thread on CPU {joj 25/7/14}
extern void waitForThreadsToFinish(UINT, THREADH*);                // {
   joj 25/7/14}
extern void closeThread(THREADH);                                  //

/*#ifdef __x86_64__
extern UINT64 rand(UINT64&);                                       // {
   joj 11/5/14}
#else*/
extern UINT rand(UINT&);                                           // {
   joj 3/1/14}
//#endif

extern int cpu64bit();                                             //
   return 1 if CPU is 64 bit
extern int cpuFamily();                                            // CPU
    family
extern int cpuModel();                                             // CPU
    model
extern int cpuStepping();                                          // CPU
    stepping
extern char *cpuBrandString();                                     // CPU
    brand string

extern int rtmSupported();                                         //
   return 1 if RTM supported (restricted transactional memory)
extern int hleSupported();                                         //
   return 1 if HLE supported (hardware lock elision)

extern int getCacheInfo(int, int, int &, int &, int&);            //
   getCacheInfo
extern int getCacheLineSz();                                       // get
    cache line sz
extern UINT getPageSz();                                           // get
    page size

extern void pauseIfKeyPressed();                                   //
   pause if key pressed
extern void pressKeyToContinue();                                  //
   press key to continue
extern void quit(int = 0);                                         //
   quit

//
// CommaLocale
//
class CommaLocale : public std::numpunct<char>
{
protected:
    virtual char do_thousands_sep() const { return ','; }
    virtual std::string do_grouping() const {return "\03"; }
};

extern void setCommaLocale();
extern void setLocale();

//
// performance monitoring
```

```
//

#define FIXED_CTR_RING0                         (1ULL)
#define FIXED_CTR_RING123                       (2ULL)
#define FIXED_CTR_RING0123                      (0x03ULL)

#define PERFEVTSEL_USR                          (1ULL << 16)
#define PERFEVTSEL_OS                           (1ULL << 17)
#define PERFEVTSEL_EN                           (1ULL << 22)
#define PERFEVTSEL_IN_TX                        (1ULL << 32)
#define PERFEVTSEL_IN_TXCP                      (1ULL << 33)

#define CPU_CLK_UNHALTED_THREAD_P               ((0x00 << 8) | 0x3c)        //
     mask | event
#define CPU_CLK_UNHALTED_THREAD_REF_XCLK        ((0x01 << 8) | 0x3c)        //
     mask | event
#define INST_RETIRED_ANY_P                      ((0x00 << 8) | 0xc0)        //
     mask | event
#define RTM_RETIRED_START                       ((0x01 << 8) | 0xc9)        //
     mask | event
#define RTM_RETIRED_COMMIT                      ((0x02 << 8) | 0xc9)        //
     mask | event

extern int openPMS();                           // open PMS
extern void closePMS();                         // close PMS
extern int pmversion();                         // return performance
    monitoring version
extern int nfixedCtr();                         // return # of fixed
    performance counters
extern int fixedCtrW();                         // return width of fixed
    counters
extern int npmc();                              // return # performance
    counters
extern int pmcW();                              // return width of performance
     counters

extern UINT64 readMSR(int, int);
extern void writeMSR(int, int, UINT64);

extern UINT64 readFIXED_CTR(int, int);
extern void writeFIXED_CTR(int, int, UINT64);

extern UINT64 readFIXED_CTR_CTRL(int);
extern void writeFIXED_CTR_CTRL(int, UINT64);

extern UINT64 readPERF_GLOBAL_STATUS(int);
extern void writePERF_GLOBAL_STATUS(int, UINT64);

extern UINT64 readPERF_GLOBAL_CTRL(int);
extern void writePERF_GLOBAL_CTRL(int, UINT64);

extern UINT64 readPERF_GLOBAL_OVF_CTRL(int);
extern void writePERF_GLOBAL_OVR_CTRL(int, UINT64);

extern UINT64 readPERFEVTSEL(int, int);
extern void writePERFEVTSEL(int, int, UINT64);

extern UINT64 readPMC(int, int);
extern void writePMC(int, int, UINT64);

// eof
```

A.2 helper.cpp

```
//
// helper.cpp
//
// Copyright (C) 2011 - 2015 jones@scss.tcd.ie
//
// This program is free software; you can redistribute it and/or modify it
    under
// the terms of the GNU General Public License as published by the Free
    Software Foundation;
// either version 2 of the License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful, but
    WITHOUT ANY WARRANTY;
// without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
    PARTICULAR PURPOSE.
// See the GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software Foundation
    Inc.,
// 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
//

//
// 12/01/11 first version
// 15/07/13 added performance monitoring support
// 14/09/13 linux support (needs g++ 4.8 or later)
// 09/02/14 added setCommaLocale and setLocale
// 10/05/14 added getVMUse and getMemUse
// 08/06/14 allocated commaLocale only once
// 26/07/14 added getPageSz()
// 26/10/14 added thread_local definitions
// 26/11/14 added AMALLOC and AFREE
//

//
// 05-08-15 enabling TSX instructions
//
// TSX was first supported by Haswell CPUs released Q2 2013.
// Not all Haswell or newer CPUs support TSX, you need to consult the
    Intel Ark database.
// In Aug-14, a bug was reported in the TSX implementation (occurs very
    rarely).
// Although this bug has been fixed in more recent CPUs (eg Broadwell and
    Skylake), many systems
// disable the buggy TSX instructions at boot time by loading microcode
    into the CPU.
// This is done by the BIOS or OS or both.
// On Windows the file C:/Windows/System32/mcupdate_GenuineIntel.dll is
    used to load the microcode into the CPU.
// Ubuntu doesn't update the microcode by default
// In order to experiment with TSX while waiting for a CPU with a bug free
    TSX implemenation, it is
// important to make sure that the TSX instruction set is enabled.
//
// T1700 (Xeon E3 1240 v3 CPU) requires version A10 BIOS +
    mcupdate_GenuineIntel.dll dated 12-11-2010
//

//
// NB: gcc needs flags -mrtm -mrdrnd
```

```
//

#include "stdafx.h"          // pre-compiled headers
#include <iostream>          // cout
#include <iomanip>           // setprecision
#include "helper.h"          //

#ifdef WIN32
#include <conio.h>           // _getch()
#include <psapi.h>           // GetProcessMemoryInfo
#elif __linux__
#include <termios.h>         //
#include <unistd.h>          //
#include <limits.h>          // HOST_NAME_MAX
#include <sys/utsname.h>     //
#include <fcntl.h>           // O_RDWR
#endif

using namespace std;         // cout. ...

//
// for data returned by cpuid instruction
//
struct _cd {
    UINT eax;
    UINT ebx;
    UINT ecx;
    UINT edx;
} cd;

UINT ncpu;                   // # logical CPUs {joj 25/7/14}
char *hostName = NULL;       // host name
char *osName = NULL;         // os name
char *brandString = NULL;    // cpu brand string

//
// getDateAndTime
//
void getDateAndTime(char *dateAndTime, int sz, time_t t)
{
    t = (t == 0) ? time(NULL) : 0;
#ifdef WIN32
    struct tm now;
    localtime_s(&now, &t);
    strftime(dateAndTime, sz, "%d-%b-%Y_%H:%M:%S", &now);
#elif __linux__
    struct tm *now = localtime(&t);
    strftime(dateAndTime, sz, "%d-%b-%Y_%H:%M:%S", now);
#endif
}

//
// getHostName
//
char* getHostName()
{
    if (hostName == NULL) {

#ifdef WIN32
        DWORD sz = (MAX_COMPUTERNAME_LENGTH + 1) * sizeof(char);
        hostName = (char*) malloc(sz);
        GetComputerNameA(hostName, &sz);
#elif __linux__
```

```
        size_t sz = (HOST_NAME_MAX + 1) * sizeof(char);
        hostName = (char*) malloc(sz);
        gethostname(hostName, sz);
#endif

    }
    return hostName;
}

//
// getOSName
//
char* getOSName()
{
    if (osName == NULL) {

        osName = (char*) malloc(256);    // should be large enough

#ifdef WIN32
        DWORD sz = 256;
        RegGetValueA(HKEY_LOCAL_MACHINE, "Software\\Microsoft\\Windows NT
            \\CurrentVersion", "ProductName", RRF_RT_ANY, NULL, (LPBYTE)
            osName, &sz);
#ifdef _WIN64
        strcat_s(osName, 256, " (64 bit)");
#else
        int win64;
        IsWow64Process(GetCurrentProcess(), &win64);
        strcat_s(osName, 256, win64 ? " (64 bit)" : " (32 bit)");
#endif
#elif __linux__
        struct utsname utsName;
        uname(&utsName);
        strcpy(osName, utsName.sysname);
        strcat(osName,  " ");
        strcat(osName, utsName.release);
#endif

    }
    return osName;
}

//
// is64bitExe
//
// return 1 if a 64 bit .exe
// return 0 if a 32 bit .exe
//
int is64bitExe()
{
    return sizeof(size_t) == 8;
}

//
// getPhysicalMemSz
//
UINT64 getPhysicalMemSz()
{
#ifdef WIN32
    UINT64 v;
    GetPhysicallyInstalledSystemMemory(&v);                    //
        returns KB
    return v * 1024;                                           // now
```

```
        bytes
#elif __linux__
    return (UINT64) sysconf(_SC_PHYS_PAGES)* sysconf(_SC_PAGESIZE); // NB:
        returns bytes
#endif
}

//
// getNumberOfCPUs
//
int getNumberOfCPUs()
{
#ifdef WIN32
    SYSTEM_INFO sysinfo;
    GetSystemInfo(&sysinfo );
    return sysinfo.dwNumberOfProcessors;
#elif __linux__
    return sysconf(_SC_NPROCESSORS_ONLN);
#endif
}

//
// cpu64bit
//
int cpu64bit()
{
    CPUID(cd, 0x80000001);
    return (cd.edx >> 29) & 0x01;
}

//
// cpuFamily
//
int cpuFamily()
{
    CPUID(cd, 0x01);
    return (cd.eax >> 8) & 0xff;
}

//
// cpuModel
//
int cpuModel()
{
    CPUID(cd, 0x01);
    if (((cd.eax >> 8) & 0xff) == 0x06)
        return (cd.eax >> 12 & 0xf0) + ((cd.eax >> 4) & 0x0f);
    return (cd.eax >> 4) & 0x0f;
}

//
// cpuStepping
//
int cpuStepping()
{
    CPUID(cd, 0x01);
    return cd.eax & 0x0f;
}

//
// cpuBrandString
//
char *cpuBrandString()
```

```
{
    if (brandString)
        return brandString;

    brandString = (char*) calloc(16*3, sizeof(char));

    CPUID(cd, 0x80000000);

    if (cd.eax < 0x80000004) {
        strcpy_s(brandString, 16*3, "unknown");
        return brandString;
    }

    for (int i = 0; i < 3; i++) {
        CPUID(cd, 0x80000002 + i);
        UINT *p = &cd.eax;
        for (int j = 0; j < 4; j++, p++) {
            for (int k = 0; k < 4; k++ ) {
                brandString[i*16 + j*4 + k] = (*p >> (k * 8)) & 0xff;
            }
        }
    }
    return brandString;
}

//
// rtmSupported (restricted transactional memory)
//
// NB: VirtualBox returns 0 even if CPU supports RTM?
//
int rtmSupported()
{
    CPUIDEX(cd, 0x07, 0);
    return (cd.ebx >> 11) & 1;      // test bit 11 in ebx
}

//
// hleSupported (hardware lock elision)
//
// NB: VirtualBox returns 0 even if CPU supports HLE??
//
int hleSupported()
{
    CPUIDEX(cd, 0x07, 0);
    return (cd.ebx >> 4) & 1;       // test bit 4 in ebx
}

//
// look for L1 cache line size (see Intel Application note on CPUID
    instruction)
//
int lookForL1DataCacheInfo(int v)
{
    if (v & 0x80000000)
        return 0;

    for (int i = 0; i < 4; i++) {
        switch (v & 0xff) {
        case 0x0a:
        case 0x0c:
        case 0x10:
            return 32;
        case 0x0e:
```

```
        case 0x2c:
        case 0x60:
        case 0x66:
        case 0x67:
        case 0x68:
            return 64;
        }
        v >>= 8;
    }
    return 0;
}

//
// getL1DataCacheInfo
//
int getL1DataCacheInfo()
{
    CPUID(cd, 2);

    if ((cd.eax & 0xff) != 1) {
        cout << "unrecognised cache type: default L 64" << endl;
        return 64;
    }

    int sz;

    if ((sz = lookForL1DataCacheInfo(cd.eax & ~0xff)))
        return sz;
    if ((sz = lookForL1DataCacheInfo(cd.ebx)))
        return sz;
    if ((sz = lookForL1DataCacheInfo(cd.ecx)))
        return sz;
    if ((sz = lookForL1DataCacheInfo(cd.edx)))
        return sz;

    cout << "unrecognised cache type: default L 64" << endl;
    return 64;
}

//
// getCacheInfo
//
int getCacheInfo(int level, int data, int &l, int &k, int &n)
{
    CPUID(cd, 0x00);
    if (cd.eax < 4)
        return 0;
    int i = 0;
    while (1) {
        CPUIDEX(cd, 0x04, i);
        int type = cd.eax & 0x1f;
        if (type == 0)
            return 0;
        int lev = ((cd.eax >> 5) & 0x07);
        if ((lev == level) & (((data == 0) & (type = 2)) || ((data == 1) &
            (type == 1))))
            break;
        i++;
    }
    k = ((cd.ebx >> 22) & 0x03ff) + 1;
    int partitions = ((cd.ebx) >> 12 & 0x03ff) + 1;
    n = cd.ecx + 1;
    l = (cd.ebx & 0x0fff) + 1;
```

```
    return partitions == 1;
}

//
// getDeterministicCacheInfo
//
int getDeterministicCacheInfo()
{
    int type, ways, partitions, lineSz = 0, sets;
    int i = 0;
    while (1) {
        CPUIDEX(cd, 0x04, i);
        type = cd.eax & 0x1f;
        if (type == 0)
            break;
        cout << "L" << ((cd.eax >> 5) & 0x07);
        cout << ((type == 1) ? " D" : (type == 2) ? " I" : " U");
        ways = ((cd.ebx >> 22) & 0x03ff) + 1;
        partitions = ((cd.ebx) >> 12 & 0x03ff) + 1;
        sets = cd.ecx + 1;
        lineSz = (cd.ebx & 0x0fff) + 1;
        cout << " " << setw(5) << ways*partitions*lineSz*sets/1024 << "K"
            << " L" << setw(3) << lineSz << " K" << setw(3) << ways << " N
            " << setw(5) << sets;
        cout << endl;
        i++;
    }
    return lineSz;
}

//
// getCacheLineSz
//
int getCacheLineSz()
{
    CPUID(cd, 0x00);
    if (cd.eax >= 4)
        return getDeterministicCacheInfo();
    return getL1DataCacheInfo();
}

//
// getPageSz
//
UINT getPageSz()
{
#ifdef WIN32
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    return si.dwPageSize;
#elif __linux__
    return sysconf(_SC_PAGESIZE);
#endif
}

//
// getWallClockMS
//
UINT64 getWallClockMS()
{
#ifdef WIN32
    return (UINT64) clock() * 1000 / CLOCKS_PER_SEC;
#elif __linux__
```

```
    struct timespec t;
    clock_gettime(CLOCK_MONOTONIC, &t);
    return t.tv_sec*1000 + t.tv_nsec / 1000000;
#endif
}

//
// setThreadCPU
//
void createThread(THREADH *threadH, WORKERF, void *arg)
{
#ifdef WIN32
    *threadH = CreateThread(NULL, 0, worker, arg, 0, NULL);
#elif __linux__
    pthread_create(threadH, NULL, worker, arg);
#endif
}

//
// runThreadOnCPU
//
void runThreadOnCPU(UINT cpu)
{
#ifdef WIN32
    SetThreadAffinityMask(GetCurrentThread(), 1 << cpu);
#elif __linux__
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(cpu, &cpuset);
    pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &cpuset);
#endif
}

//
// closeThread
//
void closeThread(THREADH threadH)
{
#ifdef WIN32
    CloseHandle(threadH);
#elif __linux__
    // nothing to do
#endif
}

//
// waitForThreadsToFinish
//
void waitForThreadsToFinish(UINT nt, THREADH *threadH)
{
#ifdef WIN32
    WaitForMultipleObjects(nt, threadH, true, INFINITE);
#elif __linux__
    for (UINT thread = 0; thread < nt; thread++)
        pthread_join(threadH[thread], NULL);
#endif
}

//
// Processor monitoring support (PMS)
//
// NB: see Intel Performance Counter Monitor v2.5.1
// NB: simplified for HLE and RTM performance measurement
```

```
//
// NB: FIXED_CTR0 counts instructions retired (see Vol 3C 35-17)
// NB: FIXED_CTR1 counts unhalted core cycles
// NB: FIXED_CTR2 counts unhalted reference cycles
//

#ifdef WIN32

typedef BOOL (WINAPI *_InitializeOls) ();
typedef VOID (WINAPI *_DeinitializeOls) ();

typedef DWORD (WINAPI *_Rdmsr) (DWORD index, PDWORD eax, PDWORD edx);
typedef DWORD (WINAPI *_Wrmsr) (DWORD index, DWORD eax, DWORD edx);

_InitializeOls InitializeOls = NULL;
_DeinitializeOls DeinitializeOls = NULL;

_Rdmsr Rdmsr = NULL;
_Wrmsr Wrmsr = NULL;

HMODULE hModule = NULL;

//
// openPMS
//
// to get the following code to work with Windows need to do the following
    :
//
// for 64bit exes, make sure WinRing0x64.dll and WinRing0x64.sys are
    placed in the same directory
// for 32bit exes, make sure WinRing0.dll and WinRing0.sys or WinRing0x64.
    sys (for 32 and 64 bit windows repspectively) are placed in the same
    directory
// exe needs to have root access eg. Run Visual Studio as Administrator
//
int openPMS()
{
    if ((hModule = LoadLibrary(is64bitExe() ? _T("WinRing0x64.dll") : _T("
        WinRing0.dll"))) == NULL)
        goto err;

    InitializeOls = (_InitializeOls) GetProcAddress(hModule, "
        InitializeOls");
    DeinitializeOls = (_DeinitializeOls) GetProcAddress(hModule, "
        DeinitializeOls");

    Rdmsr = (_Rdmsr) GetProcAddress(hModule, "Rdmsr");
    Wrmsr = (_Wrmsr) GetProcAddress(hModule, "Wrmsr");

    if (InitializeOls())
        return 1;

err:

    int win64;
    IsWow64Process(GetCurrentProcess(), &win64);

    cout << "unable to access performance monitoring counters" << endl;
    cout << "make sure " << (is64bitExe() ? "WinRing0x64.dll" : "WinRing0.
        dll") << " and " << (win64 ? "WinRing0x64.sys" : "WinRing0.sys")
        << " are in the same directory as exe" << endl;
    cout << "make sure sharing.exe is run as administrator " << endl <<
        endl;
```

```
    return 0;

}

//
// closePMS
//
void closePMS()
{
    if (hModule) {
        DeinitializeOls();
        FreeLibrary(hModule);
        hModule = NULL;
    }
}

//
// readMSR
//
UINT64 readMSR(int cpu, int addr)
{
    DWORD high, low;
    DWORD_PTR oldAffinity = SetThreadAffinityMask(GetCurrentThread(), 1 <<
        cpu);
    Rdmsr(addr, &low, &high);
    SetThreadAffinityMask(GetCurrentThread(), oldAffinity);
    return ((UINT64) high << 32) | low;
}

//
// writeMSR
//
void writeMSR(int cpu, int addr, UINT64 v)
{
    DWORD_PTR oldAffinity = SetThreadAffinityMask(GetCurrentThread(), 1 <<
        cpu);
    Wrmsr(addr, (DWORD) v, v >> 32);
    SetThreadAffinityMask(GetCurrentThread(), oldAffinity);
}

#elif __linux__

int *fd;

//
// openPMS
//
// to get the following code to work with linux need to do the following:
//
// auto load msr driver on boot by adding msr to /etc/modules
// performance counters accessed by reading from and writing to /dev/cpu/n
    /msr where n is the cpu number
// need to have root access eg. $sudo eclipse
//
// did try the following, but it didn't work
//
// created a group called msr: $sudo groupadd msr
// added user to group: $sudo usermod -a -G msr user
// added the following code to /etc/rc.local so that after boot the /dev/
    cpu/n/msr files should
// be able to be read and written by users belonging to the msr group
//
```

```
// i=0
// ncpus=`cat /proc/cpuinfo | grep processor | wc -l`
// while [ $i -lt 8 ]
// do
//   chown :msr /dev/cpu/$i/msr
//   chmod ug+rwx /dev/cpu/$i/msr
//   i=`expr $i + 1`
// done
/

//
// openPMS
//
int openPMS()
{
    char fn[32];
    int err = 0;

    ncpu = getNumberOfCPUs();

    fd = (int*) calloc(1, ncpu*sizeof(int));

    for (UINT i = 0; i < ncpu; i++) {
        sprintf(fn, "/dev/cpu/%d/msr", i);
        if ((fd[i] = open(fn, O_RDWR)) == -1) {
            cout << "unable to open " << fn << endl;
            err = 1;
        }
    }

    if (err) {
        cout << endl;
        cout << "make sure the msr driver is loaded by checking for file(s
            ) /dev/cpu/0/msr, /dev/cpu/1/msr ..." << endl;
        cout << "autoload the msr driver on boot by adding msr to /etc/
            modules" << endl;
        cout << "make sure program is run as root" << endl;
    }
    return err == 0;
}

//
// closePMS
//
void closePMS()
{
    for (UINT i = 0; i < ncpu; i++) {
        if (fd[i] != -1)
            close(fd[i]);
    }
}

//
// readMSR
//
// check result returned by write to avoid a gcc warn_unused_result
//
UINT64 readMSR(int cpu, int addr)
{
    UINT64 msr = 0;
    if (fd[cpu] != -1) {
        lseek(fd[cpu], addr, SEEK_SET);
        if (read(fd[cpu], &msr, sizeof(msr)) != sizeof(msr))
```

```
            cout << "Warning: unable to readMSR(" << cpu << ", " << addr
                << ")" << endl;
    }
    return msr;
}

//
// writeMSR
//
// check result returned by write to avoid a gcc warn_unused_result
//
void writeMSR(int cpu, int addr, UINT64 v)
{
    if (fd[cpu] != -1) {
        lseek(fd[cpu], addr, SEEK_SET);
        if (write(fd[cpu], &v, sizeof(v)) != sizeof(v))
            cout << "Warning: unable to writeMSR(" << cpu << ", " << addr
                << ", " << v << ")" << endl;
    }
}

#endif

//
// pmversion
//
int pmversion()
{
    CPUID(cd, 0x0a);
    return cd.eax & 0xff;
}

//
// nfixedctr
//
int nfixedCtr()
{
    CPUID(cd, 0x0a);
    return cd.edx & 0x1f;
}

//
// fixedctrw
//
int fixedCtrW()
{
    CPUID(cd, 0x0a);
    return (cd.edx >> 5) & 0xff;
}

//
// npmc
//
int npmc()
{
    CPUID(cd, 0x0a);
    return (cd.eax >> 8 ) & 0xff;
}

//
// pmcW
//
int pmcW()
```

```
{
    CPUID(cd, 0x0a);
    return (cd.eax >> 16) & 0xff;
}

//
// readFIXED_CTR
//
UINT64 readFIXED_CTR(int cpu, int n)
{
    return readMSR(cpu, 0x0309 + n);
}

//
// writeFIXED_CTR
//
void writeFIXED_CTR(int cpu, int n, UINT64 v)
{
    return writeMSR(cpu, 0x0309 + n, v);
}

//
// readFIXED_CTR_CTRL
//
UINT64 readFIXED_CTR_CTRL(int cpu)
{
    return readMSR(cpu, 0x038d);
}

//
// writeFIXED_CTR_CTRL
//
void writeFIXED_CTR_CTRL(int cpu, UINT64 v)
{
    return writeMSR(cpu, 0x038d, v);
}

//
// readPERF_GLOBAL_STATUS
//
UINT64 readPERF_GLOBAL_STATUS(int cpu)
{
    return readMSR(cpu, 0x038e);
}

//
// writePERF_GLOBAL_STATUS
//
void writePERF_GLOBAL_STATUS(int cpu, UINT64 v)
{
    return writeMSR(cpu, 0x038e, v);
}

//
// readPERF_GLOBAL_CTRL
//
UINT64 readPERF_GLOBAL_CTRL(int cpu)
{
    return readMSR(cpu, 0x038f);
}

//
// writePERF_GLOBAL_CTRL
```

```
//
void writePERF_GLOBAL_CTRL(int cpu, UINT64 v)
{
    return writeMSR(cpu, 0x038f, v);
}

//
// readPERF_GLOBAL_OVR_CTRL
//
UINT64 readPERF_GLOBAL_OVR_CTRL(int cpu)
{
    return readMSR(cpu, 0x0390);
}

//
// writePERF_GLOBAL_OVR_CTRL
//
void writePERF_GLOBAL_OVR_CTRL(int cpu, UINT64 v)
{
    return writeMSR(cpu, 0x0390, v);
}

//
// readPERFEVTSEL
//
UINT64 readPERFEVTSEL(int cpu, int n)
{
    return readMSR(cpu, 0x186 + n);
}

//
// writePERFEVTSEL
//
void writePERFEVTSEL(int cpu, int n, UINT64 v)
{
    return writeMSR(cpu, 0x186 + n, v);
}

//
// readPMC
//
UINT64 readPMC(int cpu, int n)
{
    return readMSR(cpu, 0xc1 + n);
}

//
// writePMC
//
void writePMC(int cpu, int n, UINT64 v)
{
    return writeMSR(cpu, 0xc1 + n, v);
}

//
// pauseIfKeyPressed
//
void pauseIfKeyPressed()
{
#ifdef WIN32
    if (_kbhit()) {
        if (_getch() == ' ') {
            cout << endl << endl << "PAUSED - press key to continue";
```

```
            _getch();
            cout << endl;
        }
    }
#elif __linux__

#endif
}

//
// pressKeyToContinue
//
void pressKeyToContinue()
{
#ifdef WIN32
    cout << endl << "Press any key to continue...";
    _getch();
#elif __linux__
    termios old, input;
    tcgetattr(fileno(stdin), &old);              // save settings
    input = old;                                 // make new settings same
        as old settings
    input.c_lflag &= ~(ICANON | ECHO);           // disable buffered i/o
        and echo
    tcsetattr(fileno(stdin), TCSANOW, &input);   // use these new terminal
        i/o settings now
    puts("Press any key to continue...");
    getchar();
    tcsetattr(fileno(stdin), TCSANOW, &old);
#endif
}

//
// quit
//
void quit(int r)
{
#ifdef WIN32
    cout << endl << "Press key to quit...";
    _getch();   // stop DOS window disappearing prematurely
#endif
    exit(r);
}

/*#ifdef __x86_64__

//
// rand
//
// due to George Marsaglia (google "xorshift wiki")
//
UINT64 rand(UINT64 &r)
{
    r ^= r >> 12;   // a
    r ^= r << 25;   // b
    r ^= r >> 27;   // c
    return r * 2685821657736338717LL;
}

#else*/

//
// rand
```

```
//
// Parker & Miller, "Random Number Generators: Good ones are hard to find
    ", CACM Vol 311 No 10 Oct-88
//
// 31 bits
//
UINT rand(UINT &r)
{
    //return r = (UINT64) r * 16807 % 2147483647;   // 31 bits
    //return r = (UINT64) r * 48271 % 2147483647;   // 31 bits
    return r = (UINT64) r * 279470273 % 4294967291; // almost 32 bits
}

//#endif

locale *commaLocale = NULL;

//
// setCommaLocale
//
void setCommaLocale()
{
    if (commaLocale == NULL)
        commaLocale = new locale(locale(), new CommaLocale());
    cout.imbue(*commaLocale);
}

//
// setLocale
//
void setLocale()
{
    cout.imbue(locale());
}

//
// getVMUse {joj 10/5/14}
//
size_t getVMUse()
{
    size_t r = 0;

#ifdef WIN32

    HANDLE hProcess;
    PROCESS_MEMORY_COUNTERS pmc;

    if ((hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
        PROCESS_VM_READ, FALSE, GetCurrentProcessId()))) {
        if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc)))
            r = pmc.PagefileUsage;
        CloseHandle(hProcess);
    }

#elif __linux__

    UINT64 vmuse;
    FILE* fp;

    if ((fp = fopen("/proc/self/statm", "r")) != NULL) {
        if (fscanf(fp, "%llu", &vmuse) == 1)
            r = vmuse * sysconf(_SC_PAGESIZE);
        fclose(fp);
```

```
    }

#endif

    return r;
}

//
// getMemUse {joj 10/5/14}
//
size_t getMemUse()
{
    size_t r = 0;

#ifdef WIN32

    HANDLE hProcess;
    PROCESS_MEMORY_COUNTERS pmc;

    if ((hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
        PROCESS_VM_READ, FALSE, GetCurrentProcessId()))) {
        if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc)))
            r = pmc.WorkingSetSize;
        CloseHandle(hProcess);
    }

#elif __linux__

    UINT64 memuse;
    FILE* fp;

    if ((fp = fopen("/proc/self/statm", "r")) != NULL) {

        if (fscanf(fp, "%*s%\llu", &memuse) == 1)
            r = memuse * sysconf(_SC_PAGESIZE);

        fclose(fp);
    }

#endif

    return r;
}

// eof
```

A.3 main.cpp

```
#include "stdafx.h"                                  // pre-compiled headers
#include <iostream>                                  // cout
#include <iomanip>                                   // setprecision
#include "helper.h"                                  //

using namespace std;                                 // cout

#define K           1024                             //
#define GB          (K*K*K)                          //
#define NOPS        65536                             //
#define NSECONDS    10                                // run each test for
    NSECONDS


#define MAX         8
unsigned int NSPLIT = 32;
#define NTAG        4096
#define TAG(a)      tag[((UINT64) (a) >> 3) % NTAG]

#define TRANSACTION         0                        // mode
#define LOCK                1                        //


UINT64 *aborts;                                      // for counting aborts


#define COUNTER64                                    // comment for 32 bit
    counter



#ifdef COUNTER64
#define VINT    UINT64                               //  64 bit counter
#else
#define VINT    UINT                                 //  32 bit counter
#endif

#define ALIGNED_MALLOC(sz, align) _aligned_malloc(sz, align)

#ifdef FALSEkeyRAnge
#define GINDX(n)    (g+n)                            //
#else
#define GINDX(n)    (g+n*lineSz/sizeof(VINT))        //
#endif

//
// OPTYP
//
// 0:inc
// 1:InterlockedIncrement
// 2:InterlockedCompareExchange
// 3:RTM (restricted transactional memory)
//

#define OPTYP       lock_type                                // set op type


#if OPTYP == 5
volatile int lock_var = 0;
#define OPSTR       "Test, Test and Set Lock"
#define ACQUIRE()   {
```

```
    \
                        do {
                            \
                            while(lock_var == 1) {
                                \
                                _mm_pause();
                                    \
                            }
                                \
                        } while(InterlockedCompareExchange(&lock_var, 1,
                            0) == 1);                   \
                    }
#define RELEASE()   lock_var = 0




#elif OPTYP == 8
volatile int lock_var = 0;
#define OPSTR       "Test, Test and Set Lock with HLE"
#define ACQUIRE()   {
    \
                        do {
                            \
                            while(lock_var == 1) {
                                \
                                _mm_pause();
                                    \
                            }
                                \
                        } while(_InterlockedExchange_HLEAcquire(&lock_var,
                            1) == 1);               \
                    }
#define RELEASE()   _Store_HLERelease(&lock_var, 0);

#elif OPTYP == 9
volatile int lock_var = 0;
#define OPSTR       "Transactional Memory with Test and Test and Set Lock
    as fallback"
#define ACQUIRE_LOCK()  {
    \
                            do {
                                \
                                while(lock_var == 1) {
                                    \
                                    _mm_pause();
                                        \
                                }
```

```
                                \
                            } while(InterlockedCompareExchange(&lock_var,
                                1, 0) == 1);                    \
                        }
#define RELEASE_LOCK()    lock_var = 0

#define ACQUIRE()

    \
                            unsigned int transaction = 1;

                                \
                            unsigned int status = 0;

                                \
                            unsigned int attempts = MAXATTEMPT;
                                                                \
                            while(1) {

                                \
                                while(lock_var) {

                                    \
                                    _mm_pause();

                                        \
                                }

                                    \
                                if(transaction == 1) {

                                    \
                                    status = _xbegin();

                                        \
                                } else {

                                    \
                                    ACQUIRE_LOCK();

                                        \
                                    nlock++;

                                        \
                                    status = _XBEGIN_STARTED;

                                        \
                                }

                                    \
                                if(status == _XBEGIN_STARTED) {
                                                                \
                                    if(_xtest() && lock_var) _xabort(0x01)
                                        ;                               \


#define RELEASE()

    \
                                    if(_xtest()) _xend();

                                        \
```

```
                else RELEASE_LOCK();                    \
                break;                                  \
            } else {                                    \
                nabort++;                               \
                if (attempts > 0) {                     \
                    attempts--;                         \
                } else transaction = 0;                 \
                while(lock_var) {                       \
                    _mm_pause();                        \
                }                                       \
                unsigned short wait = attempts;         \
                while(wait--);                          \
            }                                           \
        }                                               \
    }


unsigned short bit_shift = 0;

struct Node{
    unsigned int volatile data;
    Node* volatile next;
};

void push(Node* node, Node* &stack) {
    node -> next = stack;
    stack = node;
}

Node* pop(Node* &stack, unsigned int x) {
    Node* temp = stack;
    stack = stack -> next;
    temp -> next = NULL;
    temp -> data = x;
    return temp;
```

```
}

inline bool isEmpty(Node* stack) {
    return stack ? false : true;
}

void removeAll (Node* volatile &node) {
    Node* curr;
    while(node) {
        curr = node;
        node = curr->next;
        free(curr);
    }
}

Node* init(unsigned int x) {
    Node* temp = new Node();
    temp -> next = NULL;
    temp -> data = x;
    return temp;
}



#if OPTYP < 10
int add(Node* volatile &node, Node* newNode, unsigned long long& nlock,
    unsigned long long& nabort, unsigned short MAXATTEMPT) {
    ACQUIRE();
    Node* volatile curr = node;
    Node* volatile prev = NULL;
    if(!curr) {
        node = newNode;
#if OPTYP == 9
        if(_xtest()) _xend();
        else RELEASE_LOCK();
#else
        RELEASE();
#endif
        return 1;
    }

    if(curr -> data > newNode -> data) {
        newNode -> next = node;
        node = newNode;
#if OPTYP == 9
        if(_xtest()) _xend();
        else RELEASE_LOCK();
#else
        RELEASE();
#endif
        return 1;
    }


    do {
        if(curr -> data > newNode -> data) {
            prev -> next = newNode;
            newNode -> next = curr;
#if OPTYP == 9
            if(_xtest()) _xend();
            else RELEASE_LOCK();
#else
```

```
            RELEASE();
#endif
            return 1;
        }
        prev = curr;
        curr = curr -> next;
    } while(curr && curr -> data != newNode -> data);
    RELEASE();
    return 0;
}

#else

int add(Node* volatile &node, Node* newNode, unsigned long long& nlock,
    unsigned long long& nabort, unsigned short MAXATTEMPT, unsigned int&
    split_length)
{
    int mode = TRANSACTION;
    unsigned int attempts = MAXATTEMPT;
    unsigned int status = 0;
    unsigned int split = 0;
    unsigned long long save = 0;
retry:

    Node* volatile pp = node;                                   //head of list
        that is passed in
    Node* volatile p;

    if(mode == TRANSACTION) {

        UINT64 myTag = TAG(pp);

nextsplit:

        int backoff = 0;
        int cnt = 0;

        while (1) {

            while(lock_var) {
                _mm_pause();
            }

            status = _xbegin();

            if (status == _XBEGIN_STARTED) {
                if(split == 1) pp = (Node*) save;

                if  (lock_var)
                    _xabort(0xA0);
                if (myTag != TAG(pp)) {
                    split = 0;
                    _xabort(0xA1);
                }

                p = pp->next;

                if(!p) {
                    pp -> next = newNode;
                    _xend();
                    return 1;
                }
```

```
                if(p -> data > newNode -> data) {
                    newNode -> next = p;
                    pp -> next = newNode;
                    _xend();
                    return 1;
                }

                do {
                    if (p->data > newNode->data) {
                        pp -> next = newNode;
                        newNode -> next = p;
                        _xend();
                        return 1;
                    }
                    pp = p;
                    if (++cnt >= split_length) {
                        myTag = TAG(pp);
                        save = (unsigned long long) pp;
                        _xend();
                        split = 1;

                        split_length += 4;                          //
                            dynamic tuner
                        if(split_length > 32) split_length = 32;

                        goto nextsplit;
                    }
                    p = p->next;
                } while (p && p -> data != newNode -> data);

                _xend();

                return 0;

            } else {    // here if transaction aborts
                ++nabort;
                if (lock_var) {
                    do {
                        _mm_pause();
                    } while (lock_var);
                }
                if (--attempts <= 0) {
                    mode = LOCK;
                    goto retry;
                }
                unsigned short wait = attempts;
                while(wait--);

                split_length >>= 1;                                 //dynamic
                    tuner
                if(split_length < 8) split_length = 8;

                if(split == 1) goto nextsplit;
        }
        }
    } else { // LOCK
        do {
            do {
                _mm_pause();
            } while (lock_var);
        } while (InterlockedExchange(&lock_var, 1));

        p = pp -> next;
```

```
                if(!p) {
                    pp -> next = newNode;
                    lock_var = 0;
                    return 1;
                }

                if(p -> data > newNode -> data) {
                    newNode -> next = p;
                    pp -> next = newNode;
                    lock_var = 0;
                    return 1;
                }

                do {
                    if (p->data > newNode->data) {
                        pp -> next = newNode;
                        newNode -> next = p;
                        lock_var = 0;
                        return 1;
                    }
                    pp = p;
                    p = p->next;
                } while (p && p -> data != newNode -> data);

                lock_var = 0;

                return 0;

    }
}
#endif


bool search (Node* node, unsigned int x) {
    Node* curr = node;

    while (curr) {
        if ((curr -> data) == x) {
            return true;
        }

        if ((curr -> next) == NULL) {
            return false;
        }
        curr = curr -> next;
    }

    return false;
}



#if OPTYP < 10
Node* remove_node (Node* volatile &node, unsigned int x, unsigned long
    long& nlock, unsigned long long& nabort, unsigned short& MAXATTEMPT) {
    ACQUIRE();
    Node*volatile* curr = &node;          //node is the head of the list
    while (*curr != NULL) {
        if ((*curr)->data == x) {
            Node* temp = *curr;
            *curr = (*curr)->next;
#if OPTYP == 9
```

```
            if(_xtest()) _xend();
            else RELEASE_LOCK();
#else
            RELEASE();
#endif
            return temp;
        }
        curr = &(*curr)->next;
    }
    RELEASE();
    return NULL;
}

#else
Node* remove_node (Node* volatile &node, unsigned int x, unsigned long
    long& nlock, unsigned long long& nabort, unsigned short& MAXATTEMPT,
    unsigned int& split_length)
{
    int mode = TRANSACTION;
    unsigned int attempts = MAXATTEMPT;
    unsigned int status = 0;
    unsigned int split = 0;
    unsigned long long save = 0;
retry:

    Node* volatile pp = node;                          //node is the head
        of the list
    Node* volatile p;

    if(mode == TRANSACTION) {

        UINT64 myTag = TAG(pp);

nextsplit:
        int backoff = 0;
        int cnt = 0;

        while (1) {

            while(lock_var) {
                _mm_pause();
            }

            status = _xbegin();

            if (status == _XBEGIN_STARTED) {
                if(split == 1) pp = (Node*) save;

                if  (lock_var)
                    _xabort(0xA0);
                if (myTag != TAG(pp))
                    _xabort(0xA1);

                p = pp -> next;

                while (p) {
                    if (p -> data >= x) {
                        if (p -> data == x) {
                            pp -> next = p -> next;
                            TAG(p)++;
                            _xend();
                            return p;
                        }
```

```
                    break;
                }
                pp = p;
                if (++cnt >= split_length) {
                    myTag = TAG(pp);
                    _xend();

                    split_length += 4;                          //
                        dynamic tuner
                    if(split_length > 32) split_length = 32;

                    goto nextsplit;
                }
                p = p -> next;
            }

            _xend();
            return NULL;

        } else {    // here if transaction aborts
            ++nabort;
            if (lock_var) {
                do {
                    _mm_pause();
                } while (lock_var);
            }
            if (--attempts <= 0) {
                mode = LOCK;
                goto retry;
            }
            unsigned short wait = attempts;
            while(wait--);

            split_length >>= 1;                         //dynamic
                tuner
            if(split_length < 8) split_length = 8;

            if(split == 1) goto nextsplit;
        }
    }

} else {    // LOCK
    ++nlock;
    while (InterlockedExchange(&lock_var, 1)) {
        do {
            _mm_pause();
        } while (lock_var);
    }

    p = pp -> next;

    while (p) {
        if (p -> data >= x) {
            if (p -> data == x) {
                pp -> next = p -> next;
                TAG(p)++;
                lock_var = 0;
                return p;
            }
            break;
        }
        pp = p;
        p = p -> next;
```

```
        }

        lock_var = 0;
        return NULL;

    }
}

#endif




int pre_add(Node* volatile &node, Node* newNode) {
    Node* volatile curr = node;
    Node* volatile prev = NULL;
    if(!curr) {
        node = newNode;
        return 1;
    }

    if(curr -> data > newNode -> data) {
        newNode -> next = node;
        node = newNode;
        return 1;
    }


    do {
        if(curr -> data > newNode -> data) {
            prev -> next = newNode;
            newNode -> next = curr;
            return 1;
        }
        prev = curr;
        curr = curr -> next;
    } while(curr && curr -> data != newNode -> data);
    return 0;
}



void preFill(int size, Node* volatile &head) {

    unsigned int temp = 1;
    unsigned long long lock = 0;
    unsigned long long abort = 0;
    Node* pre;
    unsigned short max = 8;
    unsigned int split = 8;

    while(temp < size) {
        pre = init(temp);
        pre_add(head, pre);
        temp +=2;
    }

}



//testing only
```

```
void print (Node* node) {
    Node* curr = node;

    while (curr) {
        printf("%i\n", curr -> data);
        curr = curr -> next;
    }
}
```

```
UINT64 tstart;                                  // start of test in ms
int keyRAnge;                                    // % keyRAnge
int lineSz;                                     // cache line size
int maxThread;                                  // max # of threads

THREADH *threadH;                               // thread handles
UINT64 *ops;                                    // for ops per thread

unsigned long long* locks;

typedef struct {
    int keyRAnge;                                // keyRAnge
    int nt;                                     // # threads
    UINT64 rt;                                  // run time (ms)
    UINT64 ops;                                 // ops
    UINT64 incs;                                // should be equal ops
    UINT64 aborts;
    unsigned long long locks;                                //
    unsigned long long added;
    unsigned long long removed;
    unsigned long long list_size;
    unsigned int static_repeat;
} Result;

Result *r;                                      // results
UINT indx;                                      // results index

volatile VINT *g;                               // NB: position of
    volatile
```

```
WORKER worker(void *vthread)
{
    int thread = (int)((size_t) vthread);
    Node* stack_top = NULL;
    Node* temp;
    srand(thread);
    UINT64 n = 0;
    unsigned short MAXATTEMPT = MAX;
    unsigned int split_length = NSPLIT;

    unsigned int a = rand();

    unsigned long long nabort = 0;
    unsigned long long abort = 0;
    unsigned int res = 0;
    runThreadOnCPU(thread % ncpu);
```

```
    while (1) {



#if OPTYP < 10
        a = rand(a);
        if(a & 1) {
            isEmpty(stack_top) ? temp = init(a >> (30 - bit_shift)) : temp
                = pop(stack_top, (a >> (30 - bit_shift)));
            res = add(head, temp, abort, nabort, MAXATTEMPT);
            if(res == 0) push(temp, stack_top);
        } else {
            temp = remove_node(head, a >> (30 - bit_shift), abort, nabort,
                MAXATTEMPT);
            if(temp != NULL) push(temp, stack_top);
        }
#else
        a = rand(a);
        if(a & 1) {
            isEmpty(stack_top) ? temp = init(a >> (30 - bit_shift)) : temp
                = pop(stack_top, (a >> (30 - bit_shift)));
            res = add(head, temp, abort, nabort, MAXATTEMPT, split_length)
                ;
            if(res == 0) push(temp, stack_top);
        } else {
            temp = remove_node(head, a >> (30 - bit_shift), abort, nabort,
                MAXATTEMPT, split_length);
            if(temp != NULL) push(temp, stack_top);
        }
#endif

        n++;
        //
        // check if runtime exceeded
        //
        if ((getWallClockMS() - tstart) > NSECONDS*1000)
            break;

    }
    removeAll(stack_top);
    ops[thread] = n;
#if OPTYP == 9 || OPTYP == 10
    aborts[thread] = nabort;
    locks[thread] = abort;
#endif
    return 0;

}

//
// main
//
int main()
{
    ncpu = getNumberOfCPUs();    // number of logical CPUs
    maxThread = ncpu * 2;        // max number of threads

    //
    // get date
    //
    char dateAndTime[256];
    getDateAndTime(dateAndTime, sizeof(dateAndTime));
```

```
    //
    // console output
    //
    cout << getHostName() << " " << getOSName() << " keyRAnge " << (
        is64bitExe() ? "(64" : "(32") << "bit EXE)" ;
#ifdef _DEBUG
    cout << " DEBUG";
#else
    cout << " RELEASE";
#endif
    cout << " [" << OPSTR << "]" << " NCPUS=" << ncpu << " RAM=" << (
        getPhysicalMemSz() + GB - 1) / GB << "GB " << dateAndTime << endl;
#ifdef COUNTER64
    cout << "COUNTER64";
#else
    cout << "COUNTER32";
#endif
#ifdef FALSEkeyRAnge
    cout << " FALSEkeyRAnge";
#endif
    cout << " NOPS=" << NOPS << " NSECONDS=" << NSECONDS << " OPTYP=" <<
        OPTYP;
#ifdef USEPMS
    cout << " USEPMS";
#endif
    cout << endl;
    cout << "Intel" << (cpu64bit() ? "64" : "32") << " family " <<
        cpuFamily() << " model " << cpuModel() << " stepping " <<
        cpuStepping() << " " << cpuBrandString() << endl;
#ifdef USEPMS
    cout << "performance monitoring version " << pmversion() << ", " <<
        nfixedCtr() << " x" << fixedCtrW() << "bit fixed counters, " <<
        npmc() << " x " << pmcW() << "bit performance counters" << endl;
#endif

    //
    // get cache info
    //
    lineSz = getCacheLineSz();
    //lineSz *= 2;



    cout << endl;

    //
    // allocate global variable
    //
    // NB: each element in g is stored in a different cache line to stop
        false keyRAnge
    //
    threadH = (THREADH*) ALIGNED_MALLOC(maxThread*sizeof(THREADH), lineSz)
        ;             // thread handles
    ops = (UINT64*) ALIGNED_MALLOC(maxThread*sizeof(UINT64), lineSz);
                          // for ops per thread

#if OPTYP == 10 || OPTYP == 9
    aborts = (UINT64*) ALIGNED_MALLOC(maxThread*sizeof(UINT64), lineSz);
                        // for counting aborts
    locks = (UINT64*) ALIGNED_MALLOC(maxThread*sizeof(UINT64), lineSz);
#endif
```

```
    r = (Result*) ALIGNED_MALLOC(32*maxThread*sizeof(Result), lineSz);
                            // for results
    memset(r, 0, 5*maxThread*sizeof(Result));
                                                    // zero

    indx = 0;


    //
    // use thousands comma separator
    //
    setCommaLocale();

    //
    // header
    //
    cout << "    " << "List Size";
    cout << setw(3) << "nt";
    cout << setw(6) << "rt";
    cout << setw(16) << "ops";
    cout << setw(16) << "ops/s";
#if OPTYP == 10 || OPTYP == 9
    cout << setw(18) << "Locks taken";
    cout << setw(6) << "%";
    cout << setw(18) << "Aborts";
    cout << setw(18) << "Aborts per Op";
   // cout << setw(12) << "Static";
#endif
    cout << endl;
    cout << "    " << "--------";                    // keyRAnge
    cout << setw(4) << "--";          // nt
    cout << setw(6) << "--";          // rt
    cout << setw(16) << "---";        // ops
    cout << setw(16) << "---";         // rel
#if OPTYP == 10 || OPTYP == 9
    cout << setw(15) << "------";
    cout << setw(10) << "--";
    cout << setw(16) << "------";
    cout << setw(18) << "-----";
    //cout << setw(12) << "----";
#endif
    cout << endl;


    bit_shift = 0;


    for(keyRAnge = 16;keyRAnge <= NOPS;keyRAnge*=4) {
        bit_shift += 2;


        for (int nt = 1; nt <= maxThread; nt*=2, indx++) {

            //
            //  zero shared memory
            //

            head = NULL;

            preFill(keyRAnge, head);

#if OPTYP >= 10
            memset((void*) tag, 0, NTAG*sizeof(UINT64));
```

```
#endif



            //
            // get start time
            //

            tstart = getWallClockMS();

            //
            // create worker threads
            //


            for (int thread = 0; thread < nt; thread++)
                createThread(&threadH[thread], worker, (void*)(size_t)
                    thread);

            //
            // wait for ALL worker threads to finish
            //
            waitForThreadsToFinish(nt, threadH);
            UINT64 rt = getWallClockMS() - tstart;



            //
            // save results and output summary to console
            //
            for (int thread = 0; thread < nt; thread++) {
                r[indx].ops += ops[thread];
                r[indx].incs += *(GINDX(thread));
#if OPTYP == 10 || OPTYP == 9
                r[indx].aborts += aborts[thread];
                r[indx].locks += locks[thread];
//                r[indx].static_repeat = MAX + 1;
#endif
            }
            r[indx].incs += *(GINDX(maxThread));
            if ((keyRAnge == 0) && (nt == 1))
                ops1 = r[indx].ops;
            r[indx].keyRAnge = keyRAnge;
            r[indx].nt = nt;
            r[indx].rt = rt;

            cout << setw(10) << keyRAnge;
            cout << setw(4) << nt;
            cout << setw(8) << fixed << setprecision(2) << (double) rt /
                1000;
            cout << setw(18) << r[indx].ops;
            cout << setw(16) << fixed << setprecision(2) << (double) r[
                indx].ops / ((double) rt / 1000);

#if OPTYP == 10 || OPTYP == 9

            cout << setw(12) << r[indx].locks;
            cout << setw(10) << fixed << setprecision(2) << (double) (r[
                indx].locks*100)/r[indx].ops;
            cout << setw(16) << r[indx].aborts;
            cout << setw(18) << fixed << setprecision(2) << (double) r[
```

```
                indx].aborts/r[indx].ops;
 //             cout << setw(12) << MAX;
#endif

            //if (r[indx].ops != r[indx].incs)
                //cout << " ERROR incs " << setw(3) << fixed <<
                    setprecision(0) << 100.0 * r[indx].incs / r[indx].ops
                    << "% effective";

            cout << endl;

            //
            // delete thread handles
            //
            for (int thread = 0; thread < nt; thread++)
                closeThread(threadH[thread]);

            removeAll(head);
#if OPTYP == 10
            head = new Node();
            head -> data = 0;
#endif
        }
    //    }

    }

    cout << endl;

    //
    // output results so they can easily be pasted into a spread sheet
        from console window
    //
    setLocale();
    cout << "Tree Size/nt/rt/ops/ops-per-s";
#if OPTYP == 10 || OPTYP == 9
    cout << "/Lock Taken/%Lock Taken";
#endif
    cout << endl;
    for (UINT i = 0; i < indx; i++) {
        cout << r[i].keyRAnge << "/" << r[i].nt << "/" << r[i].rt << "/"
            << r[i].ops << "/" << (double) r[i].ops / ((double) r[i].rt /
            1000);
#if OPTYP == 10 || OPTYP == 9
        cout << "/" << r[i].locks;
        cout << "/" << fixed << setprecision(2) << (double) (r[i].locks
            *100)/r[i].ops;
        cout << "/" << r[i].aborts;
        cout << "/" << fixed << setprecision(2) << (double) r[i].aborts/r[
            i].ops;
//        cout << "/" << r[i].static_repeat;
#endif
        cout << "/" << endl;
    }
    cout << endl;

    quit();

    return 0;

}

// eof
```

A.4 Build File

```
BIN = sharing
CC = g++
#FLAGS = -Wall -pedantic
OPTS = -Dlock_type=10
SYS_LIB = -mrtm -mrdrnd -O3 -pthread
SRC = main.cpp helper.cpp

all:
	${CC} ${FLAGS} -o ${BIN} ${SRC} ${SYS_LIB} ${OPTS}
```